Université de Mons
Faculté des Sciences
Département d'Informatique

F.R.S.-FNRS

Universiteit Antwerpen
Faculteit Wetenschappen
Departement Informatica

Active Learning of Automata with Resources

Gaëtan STAQUET

A dissertation submitted in fulfillment of the requirements of the degree of
Docteur en Sciences of the Université de Mons and
Doctor in de wetenschappen: informatica of the Universiteit Antwerpen

September 2024

Jury

Véronique Bruyère UMONS – Université de Mons	Cosupervisor
Dana Fisman Ben-Gurion University	Reviewer
Daniel Neider TU Dortmund University	Reviewer
Guillermo A. Pérez Universiteit Antwerpen	Cosupervisor
Ocan Sankur CNRS, Devine team	Reviewer
Frits W. Vaandrager Radboud Universiteit	Reviewer
Jef Wijsen UMONS – Université de Mons	President

Abstract

Computer systems are ubiquitous nowadays and it goes without saying that their correctness is of capital importance in a lot of cases. However, identifying bugs and faults in computer systems is a hard and complex task. On top of well-known methods such as unit testing, integration testing, and so on, one can apply *model checking* techniques, which formally verify that a *model* (an abstract representation of the system) behaves correctly with regards to a set of constraints. Constructing a model from a system is itself complex and may introduce errors that do not occur in the actual system. Fortunately, if the system can be modeled by an *automaton* (a state machine describing which execution is valid or invalid), one can apply *active automata learning* algorithms to automatically construct an automaton by interacting with the computer system in a black-box manner, *i.e.*, by only observing runs of the system without having access to its internal details. While the original algorithm introduced in 1987 by Dana Angluin focused on simple automata than can only use their states to determine whether an execution is valid or not, many efforts were made in recent years to learn more complex (and, thus, more expressive) families of automata that can use *resources*, such as a stack, registers, *etc.* In this thesis, we present two learning algorithms for two distinct extensions of automata (with different available resources), as well as a model checking approach for JSON documents, relying on automata learning. We divide our contributions into three axes.

Firstly, we provide a learning algorithm for a family of automata extended with a natural counter, which can be incremented or decremented along the transitions. Furthermore, it can be tested against zero, allowing different behaviors based on the current counter value. Since the counter does not have an upper bound in general, the number of pairs of a state and a counter value is potentially infinite, meaning that learning the behavior of a system requires special care. We provide a finite characterization of this behavior that can be learned by querying the system, and from which a one-counter automata can be extracted. We show that the algorithm builds a polynomial number of hypotheses in the size of this characterization but requires exponentially many interactions to do so.

Secondly, we focus on JSON documents, which can be used to store and transfer information in a way that is easily readable by a human and by a computer. More precisely, we assume that we are in a streaming context, *i.e.*, the document is received piece by piece (which happens when a document is sent via a network, for instance), and that we want to verify whether the document is valid with regards to a set of constraints, given as a *JSON schema*. The classical algorithm exploring the constraints and the document in parallel requires to keep the full document in memory in the worst case and, thus, is not always appropriate in a streaming scenario. Our new approach first learns an automaton augmented with a stack that is then abstracted and used to efficiently decide whether a document is valid, without needing to store the whole document. That is, our validation algorithm has a lower overall memory requirement, at the cost of needing more time to validate a document, as observed on experimental results.

Finally, we study automata whose resources are timers that can be used to encode timing constraints. A timer is started at some value and decreases over time. Then, when it reaches zero, a special event occurs that must be handled, similarly to interruptions in a processor. It may happen that multiple timers reach zero at the same time, or that the user provides an input exactly at the same time a timer times out. In these cases, the model has a non-deterministic behavior as the automaton may decide to process these events in any order. We study the timed behavior of such an automaton and provide conditions ensuring that any untimed behavior can be observed without this non-determinism. Finally, we give an active learning algorithm, requiring a factorial number of interactions in the

number of timers, and polynomial in the number of states. As, in practice, the number of timers remains relatively small, we claim that our algorithm can be used for real-world applications.

Résumé en français

De nos jours, les systèmes informatiques sont omniprésents et il va sans dire que leur exactitude est d'une importance capitale dans bien des cas. Cependant, identifier des bogues et des défauts dans les systèmes informatiques est une tâche dure et complexe. En plus des méthodes bien connues comme les tests unitaires, les tests d'intégration, *etc.*, des méthodes de *vérification de modèles* (*model checking*, en anglais) peuvent être appliquées. Ces méthodes vérifient formellement qu'un *modèle* (une représentation abstraite du système) se comporte correctement par rapport à un ensemble de contraintes. La construction d'un modèle à partir d'un système est en soi complexe et peut introduire des erreurs qui n'existent pas dans le système concret. Heureusement, si le système peut être modélisé par un *automate* (une machine à états décrivant les exécutions valides ou invalides du système), il est possible d'utiliser des algorithmes d'*apprentissage actif d'automates* pour automatiquement construire un automate en interagissant avec le système informatique comme une boite noire, c'est-à-dire, en ne pouvant qu'observer les exécutions du système sans avoir accès à ses détails internes. L'algorithme originel introduit en 1987 par Dana Angluin se concentrait sur des automates simples qui ne peut utiliser que leurs états pour déterminer si une exécution est valide ou non. Ces dernières années, plusieurs travaux ont été effectués afin d'apprendre des familles d'automates plus complexes (et, donc, plus expressifs), capables d'utiliser des *ressources*, tels qu'une pile, des registres, *etc.* Dans cette thèse, nous présentons deux algorithmes d'apprentissage for deux extensions distinctes d'automates (avec des ressources disponibles différentes), ainsi qu'une approche de model checking pour des documents JSON, qui se base sur l'apprentissage d'automates. Nous séparons nos contributions en trois axes.

Premièrement, nous décrivons un algorithme d'apprentissage pour une famille d'automates étendus avec un compteur naturel qui peut être incrémenté ou décrémenté le long des transitions de ces automates. De plus, il est possible de vérifier si le compteur vaut zéro, ce qui permet des comportements différents selon la valeur du compteur. Comme le compteur n'a pas de borne supérieure en général, le nombre de paires composées d'un état et d'une valeur de compteur est potentiellement infini, ce qui veut dire qu'apprendre le comportement d'un tel système nécessite une attention particulière. Nous décrivons une caractérisation finie de ce comportement qui peut être apprise en interagissant avec le système et à partir de laquelle un automate à un compteur peut être extrait. Nous montrons que l'algorithme construits un nombre polynomial d'hypothèses en la taille de cette caractérisation mais nécessite un nombre exponentiel d'interactions pour y arriver.

Deuxièmement, nous nous concentrons sur les documents JSON qui peuvent être utilisés pour stocker et transférer de l'information d'une manière facilement lisible pour un humain et un ordinateur. Plus précisément, nous supposons que nous sommes dans un context de diffusion en continu (streaming, en anglais), c'est-à-dire que le document est reçu pièce par pièce (ce qui arrive lorsque le document est transféré via un réseau, par exemple). De plus, nous voulons vérifier si le document est valide par rapport à un ensemble de contraintes, donné comme un *schéma JSON*. L'algorithme classique qui explore les contraintes et le document en parallèle nécessite de garder le document complet en mémoire dans le pire des cas et n'est donc pas toujours approprié dans un scénario de streaming. Notre nouvelle approche apprend d'abord un automate augmenté d'une pile qui est ensuite abstrait et utilisé pour décider efficacement si un document est valide, sans avoir besoin de stocker le document complet. Autrement dit, notre algorithme de validation requiert globalement moins de mémoire mais a besoin de plus de temps pour valider un document, comme le montrent nos résultats expérimentaux.

Finalement, nous étudions des automates étendus avec des minuteurs qui peuvent être utilisés pour encoder des contraintes de temps. Un minuteur est démarré à une certaine valeur et décroît avec le temps. Lorsqu'il arrive à zéro, un événement spécial se déclenche qui doit alors être traité, comme les interruptions dans un processeur, par exemple. Il peut arriver que plusieurs minuteurs arrivent à zéro en même temps ou que l'utilisateur donne une entrée exactement au même moment où un minuteur expire. Dans ces cas, le modèle a un comportement non-déterministe vu que l'automate peut décider de traiter ces événements dans n'importe quel ordre. Nous étudions le comportement temporisé d'un tel automate et donnons des conditions qui assurent que n'importe quel comportement non-temporisé peut être observé sans ce non-déterminisme. Pour finir, nous décrivons un algorithm d'apprentissage actif qui nécessite un nombre factoriel d'interactions en le nombre de minuteurs et polynomial en le nombre d'états. Comme, en pratique, le nombre de minuteurs reste relativement bas, nous prétendons que notre algorithme peut être utilisé pour des applications du monde réel.

Samenvatting in het Nederlands

Computersystemen zijn tegenwoordig alomtegenwoordig en het spreekt voor zich dat hun correctheid in veel gevallen van kapitaal belang is. Het identificeren van bugs en fouten in computersystemen is echter een moeilijke en complexe taak. Naast bekende methodes zoals unit testen, integratietesten, etc., kunnen *modelcontrole* (*model checking*, in Engels) methodes worden toegepast. Deze methodes controleren formeel of een *model* (een abstracte weergave van het systeem) zich correct gedraagt in relatie tot een set beperkingen. De constructie van een model van een systeem is op zichzelf complex en kan fouten introduceren die niet bestaan in het werkelijke systeem. Gelukkig is het, als het systeem kan worden gemodelleerd door een *automaat* (een toestandsmachine die de geldige of ongeldige uitvoeringen van het systeem beschrijft), mogelijk om *algoritmen voor het actieve leren van automaten* te gebruiken om automatisch een automaat te construeren door interactie met het computersysteem als een zwarte doos, dat wil zegen, door alleen de uitvoeringen van het systeem te kunnen observeren zonder toegang te hebben tot de interne details. Het oorspronkelijke algoritme dat in 1987 werd geïntroduceerd door Dana Angluin richtte zich op eenvoudige automaten die alleen hun toestanden konden gebruiken om te bepalen of een uitvoering een uitvoering geldig is of niet. In de afgelopen jaren is er veel werk verricht om complexere (en dus expressievere) families van automaten te leren, die *bronnen* kunnen gebruiken zoals een stack, registers, etc. In deze doctoraatsthesis presenteren we twee leeralgoritmen voor twee verschillende uitbreidingen van automaten (met verschillende beschikbare bronnen), evenals een modelcontrolebenadering voor JSON-documenten, gebaseerd op automaatleren We verdelen onze bijdragen in drie gebieden.

Eerst beschrijven we een leeralgoritme voor een familie van uitgebreide automaten met een natuurlijke teller die kan worden verhoogd of verlaagd langs de overgangen van deze automaten. Bovendien is het mogelijk om te controleren of de teller nul is, waardoor verschillende gedragingen mogelijk zijn afhankelijk van de waarde van de teller. Aangezien de teller in het algemeen geen bovengrens heeft, is het aantal paren bestaande uit een toestand en een tellerwaarde potentieel oneindig, wat betekent dat het leren van het gedrag van een dergelijk systeem speciale aandacht vereist. We beschrijven een eindige karakterisering van dit gedrag dat kan worden geleerd door interactie met het systeem en waaruit een automaat met één teller geëxtraheerd kan worden. We laten zien dat het algoritme een polynomiaal aantal aannames construeert construeert in de grootte van deze karakterisering, maar een exponentieel aantal van interacties om dit te bereiken.

Ten tweede richten we ons op JSON documenten die gebruikt kunnen worden om informatie op te slaan en over te dragen op een manier die gemakkelijk leesbaar is voor zowel mensen als computers. Specifiek gaan we ervan uit dat we ons in een streaming context bevinden, dat wil zegen, dat het document stukje bij beetje wordt ontvangen (wat bijvoorbeeld gebeurt als het document via een netwerk wordt overgedragen). Daarnaast willen we controleren of het document geldig is met betrekking tot een set van beperkingen, gegeven als een *JSON schema*. Het klassieke algoritme dat de beperkingen en het document parallel onderzoekt vereist in het slechtste geval dat het hele document in het geheugen wordt bewaard en is daarom niet altijd geschikt in een streaming scenario. Onze nieuwe aanpak leert eerst een stack augmented automaton die vervolgens wordt geabstraheerd en gebruikt om efficiënt te beslissen of een document geldig is, zonder dat het volledige document hoeft te worden opgeslagen. Met andere woorden, ons validatiealgoritme heeft in totaal minder geheugen nodig, maar heeft meer tijd nodig om een document te valideren, zoals onze experimentele resultaten laten zien.

Tenslotte bestuderen we automaten waarvan de bronnen timers zijn die kunnen orden gebruikt om tijdsbeperkingen te coderen. Een timer wordt gestart op een bepaalde waarde en neemt af in de tijd. Wanneer deze nul bereikt, vindt er een speciale gebeurtenis plaats die afgehandeld moet worden, vergelijkbaar met onderbrekingen in een processor. Het kan gebeuren dat meerdere timers tegelijkertijd nul bereiken, of dat de gebruiker een invoer geeft precies op het moment dat een timer afloopt. In deze gevallen heeft het model een niet-deterministisch gedrag omdat de automaat kan besluiten om deze gebeurtenissen in elke willekeurige volgorde te verwerken. We bestuderen het getimede gedrag van zo'n automaat en geven voorwaarden die ervoor zorgen dat elk ongetimed gedrag kan worden waargenomen zonder dit niet-determinisme. Tenslotte geven we een algoritme voor actief leren, dat een factoriaal aantal interacties in het aantal timers en een polynoom in het aantal toestanden vereist. Aangezien het aantal timers in de praktijk relatief klein blijft, beweren we dat ons algoritme kan worden gebruikt voor echte toepassingen.

Acknowledgments

Look to those who walked before to lead those who walk after.

First and foremost, I am deeply grateful to my supervisors, Véronique Bruyère and Guillermo A. Pérez, for their time, their support and all their wise advice. Working with you two has been a pleasure and an enriching experience. Thank you for helping me grow as a researcher and a human.

I would like to sincerely thank all the jury members: Dana Fisman, Daniel Neider, Ocan Sankur, Frits W. Vaandrager, and Jef Wijsen. I am very happy you agreed to review my work and this pretty long thesis.

Even though my name is the only one to appear on the cover page, this work resulted from collaborations. Hence, additionally to my supervisors, I am thankful to Bharat Garhewal and Frits W. Vaandrager for all the exchanged ideas, time spent together figuring out how to learn Mealy machines with timers, and advice to improve my writing skills.

I am very fortunate to have met wonderful office co-workers throughout these four years. Thank you, Aline, Clément, Pierre, James, Chloé, Nicolas, Christophe, and Luca, for being integral part of the life of the research group at Mons; Quentin, Kévin, Giovanni, Gauvain, Ali, Phithak, Aqeel, Christophe, Martin, and Dyna for being great office mates; to everyone I have had to pleasure to discuss with at Mons; Ritam, Tim, Kasper, Shrisha, Ramesh for the (too) few moments spent together in Antwerp; Ingrid, Frits, Bharat, Eline, Loes, Thiago, Marnix, Maris, Merlijn, Tom, Chrisoph, Ruben, and everyone from the department of Software Science at Radboud Universiteit for welcoming me during the three months of my stay; Christophe, Nicolas, Hugo, and Julien for being great interns.

I also want to express my sincerest thanks to all my friends: Clothilde, Laura, Mona, and Nelson for all the days and hours spent with you, for all the board games parties, for the vacations, for being there when I feel down, for the Tinkerbell costumes, for helping me growing into a better man; Clothilde, Laura, Mona, Mathieu, and everyone else involved in our theater plays; Lydie, Marion, Cécile, and Sophie, for still being there after all those years since high school, and for all these board game sessions with Thibault, Gauthier, and Hugo; Aline, Clément, Pierre, Horacio, James for the online breaks during the confinements; Aline, Pierre, James, Chloé, Nicolas, Christophe, Luca, and Noémie for the hours spent outside the university and playing board games; Jérémy, Camille, Victor, Pauline, Giovanni, Alexis, for our discussions while messing around in Éorzéa.

At last but not least, I am grateful to my parents and my family for always being there and supporting (in every sense of the word) me throughout the years.

To the friends lost to time, to the friends currently in my life, to the friends I have yet to meet, thank you!

Contents

List of Figures

List of Tables

List of Algorithms

Introduction 1.

In this chapter, we introduce *automata* and motivate the problem of automatically inferring an automaton from a concrete system, known as *automata learning*. We then sketch the original contributions in this thesis and describe an outline of the document structure.

Chapter contents

1.1. Context

Computer systems are ubiquitous nowadays and it goes without saying that their correctness is of capital importance in a lot of cases. A rocket that fails to compute a good trajectory, or an electronic banking system that erroneously adds or withdraws money from an account may have catastrophic consequences, be financial or lethal. We refer to [BK08, Chapter 1] for more examples and arguments on the necessity of verifying systems. However, identifying bugs and faults in computer systems is in itself a hard and complex task. On top of well-known methods such as unit testing, integration testing, and so on, one can apply *model checking* techniques, which formally verify that a *model* (an abstract representation of the system) behaves correctly with regards to a set of requirements. If it is not the case, a clear counterexample must be provided, allowing the developers to understand and fix the bugs. Furthermore, the models we want to use ought to be understandable for humans, while being analyzable fully automatically.

[BK08]: Baier et al. (2008), *Principles of model checking*

Let us illustrate this idea of abstracting a system by considering a program verifying that an algebraic expression is well-formed, in the sense that the numbers of opening and closing braces match. This can easily be abstracted with a *counter*: each time we read an opening brace, we increment the counter; each time we read a closing brace, we decrement the counter. Then, an expression is valid only if the counter is zero after processing the whole expression and it never went below zero (assuming the initial value of the counter is zero). While this works when we only consider braces, adding curly or square brackets makes the system harder to model. Instead of simply counting the number of opened braces, we have to remember the exact order of the various brackets, *e.g.*, if we saw `(`, then `{`, we want to see a `}` before a `)`. In that case, one could consider using a *stack*. We start with an empty stack, then every time we see an opening symbol (*i.e.*, a `(`, a `{`, or a `[`), we push that symbol on the stack. When we see a closing symbol, we check that the symbol at the top of the stack matches the current symbol (*e.g.*, if we see a `}`, the symbol at the top of the stack must be a `{`). If the symbols do not match

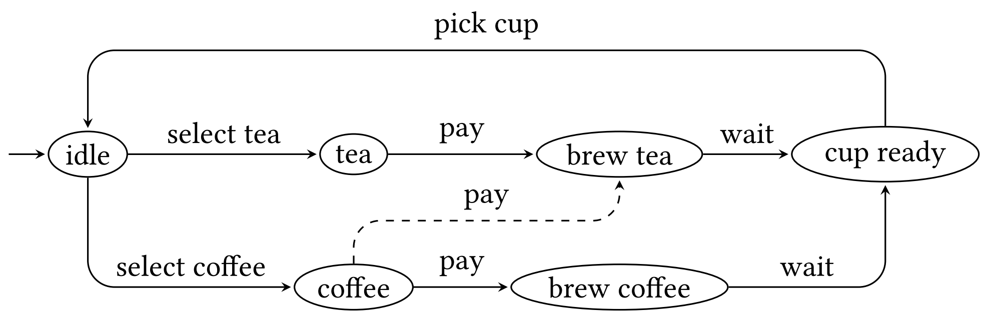


Figure 1.1: An automaton for a vending machine selling tea and coffee. The transition drawn with dashes is incorrect and shows that the system has a wrong behavior.

or if the stack is not empty after processing the whole expression, then the expression is not valid.

The considered models. For now, let us assume that the system can be modeled by an *automaton*, which is a finite state machine describing which execution is valid or invalid. It is noteworthy that many model checking algorithms exist for automata [BK08; Cla+18].

[BK08]: Baier et al. (2008), *Principles of model checking*
[Cla+18]: Clarke et al. (2018), *Handbook of Model Checking*

Figure 1.1 gives an example of an automaton that models a vending machine for tea and coffee. The automaton starts in the state *idle*, waiting for a user to interact with it. Say that a user desires to drink tea. After pressing a button, which triggers a change of state in the automaton (it is now in the state *tea*), the price is shown on the display of the vending machine. Then, the user pays the requested price (the automaton is now in *brew tea*) and has to wait for the tea to be ready and poured into a cup (the state *cup ready*). Once the cup is picked up, the automaton goes back into its idle state, waiting for the next order to process.

Observe that the automaton has an execution that is incorrect: once the user has selected a type of coffee, the machine may erroneously produce a cup of tea. The problematic arrow is drawn with dashes, to highlight it. There are thus two possibilities:

- either this wrong behavior exists in the concrete system, in which case we have identified a bug;
- or the automaton is an incorrect abstraction of the system.

It is highly important that the considered model is a good representation of the concrete system, in order to conclude that it is correct or erroneous.

Obtaining a good model. In practice, abstract models of computer systems are typically not available for legacy and for AI systems constructed from training data. Moreover, constructing a model from a system is itself complex and may introduce errors that do not occur in the actual system.

Fortunately, if the system can be modeled by an automaton, one can apply *active automata learning* algorithms to automatically construct automata by interacting with the computer system in a black box manner, *i.e.*, by only observing runs of the system without having access to its internal details.

Notably, active automata learning algorithms have been successfully used in numerous applications, for instance, for spotting bugs in implementations of major network protocols, *e.g.*, in [RP15; FJV16; Fit+17; FH17; Fit+20; Fer+21]. While we give the main ideas here, we refer to [Vaa17; HS18; Aic+18] for more-in-depth surveys and further references, and to [BBM21] for an introduction to learning algorithms.

[RP15]: Ruiter et al. (2015), "Protocol State Fuzzing of TLS Implementations"
[FJV16]: Fiterau-Brostean et al. (2016), "Combining Model Learning and Model Checking to Analyze TCP Implementations"
[Fit+17]: Fiterau-Brostean et al. (2017), "Model learning and model checking of SSH implementations"
[FH17]: Fiterau-Brostean et al. (2017), "Learning-Based Testing the Sliding Window Behavior of TCP Implementations"
[Fit+20]: Fiterau-Brostean et al. (2020), "Analysis of DTLS Implementations Using Protocol State Fuzzing"
[Fer+21]: Ferreira et al. (2021), "Prognosis: closed-box analysis of network protocol implementations"
[Vaa17]: Vaandrager (2017), "Model learning"
[HS18]: Howar et al. (2018), "Active Automata Learning in Practice - An Annotated Bibliography of the Years 2011 to 2016"
[Aic+18]: Aichernig et al. (2018), "Model Learning and Model-Based Testing"
[BBM21]: Björklund et al. (2021), "Learning algorithms"

The original learning algorithm proposed in 1987 by Dana Angluin [Ang87] focused on simple automata than can only use their states to determine whether a word is valid or not. Many efforts were made in recent years to learn more complex (and, thus, more expressive) families of automata that can use external *resources*, such as a stack, registers, *etc.* In this thesis, we present two learning algorithms for two distinct extensions of automata (with different available resources), as well as a model checking approach for JSON documents, relying on automata learning.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

We first give a more thorough account of the state of the art about active automata learning, before introducing our main contributions. Finally, we give an outline of the structure of this thesis.

1.2. Learning an automaton from a system

Let us assume we have a *system under learning* (*SUL*, for short) that can be abstracted into an automaton (using maybe some resources, as said above). There are two main ways to *learn* an automaton from the SUL.

Passive learning If one fixes a set $\mathcal{P}$ of good, positive executions and a set $\mathcal{N}$ of bad, negative executions of the SUL, it is possible to use a *passive learning algorithm* that constructs an automaton that is consistent with $\mathcal{P}$ and $\mathcal{N}$, *i.e.*, that recognizes every execution of $\mathcal{P}$ as valid, and every execution of $\mathcal{N}$ as invalid.
As, in general, there are infinitely many different executions of the SUL (as, in all generality, we cannot set a bound over the length of an execution), there must exist some executions that are neither in $\mathcal{P}$ nor in $\mathcal{N}$. Such runs can be either considered valid or invalid by the produced automaton, *i.e.*, they do not constrain the passive learning approach.
Many passive algorithms exist and can be grouped into different categories:

- Some algorithms describe the executions of $\mathcal{P} \cup \mathcal{N}$ in Boolean logic and then call a SAT solver to produce an automaton. See, for instance, [BF72; HV10].
- In a similar vein, some algorithms instead rely on SMT solving, such as [NJ13].
- Finally, some other algorithms instead build a naive automaton from $\mathcal{P}$ and $\mathcal{N}$ and then try to merge states sharing the same behavior and such that the resulting automaton remains consistent with $\mathcal{P}$ and $\mathcal{N}$, *e.g.*, [OG92].

[BF72]: Biermann et al. (1972), "On the Synthesis of Finite-State Machines from Samples of Their Behavior"
[HV10]: Heule et al. (2010), "Exact DFA Identification Using SAT Solvers"
[NJ13]: Neider et al. (2013), "Regular Model Checking Using Solver Technologies and Automata Learning"
[OG92]: Oncina et al. (1992), "Inferring regular languages in polynomial updated time"

We refer to [Nei14; BBM21] for a description of some passive learning algorithms.

[Nei14]: Neider (2014), "Applications of automata learning in verification and synthesis"
[BBM21]: Björklund et al. (2021), "Learning algorithms"

Active learning Instead of fixing the two sets $\mathcal{P}$ and $\mathcal{N}$ of executions, one can consider *querying* the SUL, *i.e.*, providing it some input sequences in order to observe the resulting output. Algorithms using this idea are called *active learning algorithms* and are structured into two distinct parts:

- a *teacher* that knows the SUL and how to interact with it, and
- a *learner* that initially knows nothing about the SUL but can *query* the teacher to gather knowledge.

We call this structure, the *Angluin's framework* [Ang87].

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

In this thesis, we solely focus on active learning algorithms. We briefly discuss L^*,[1] the learning algorithm proposed by Angluin [Ang87] that can infer "simple" automata, in the sense that they can only use their states to distinguish between valid and invalid executions. We assume that the teacher knows exactly the set of all valid executions of the SUL and can answer two types of queries:

1: The algorithm is presented with more details in Chapter 3.

- membership queries where the learner provides an execution and asks whether it should be deemed valid, and
- equivalence queries where the learner provides a hypothesis (an automaton that is conjectured to be correct with regards to the SUL) and asks whether there exists a difference in behavior between the SUL and that hypothesis.

Noteworthily, this assumes that the teacher is able to answer these queries simply by interacting with the SUL. That is, we assume that the teacher (and, hence, the learner) does not know the internal details of the SUL, *i.e.*, the SUL is seen as a *black box*.

We now give several research directions that were explored in the furrow of [Ang87].

Learning with approximate queries. In many practical situations, the teacher may not be able to perfectly answer every query. In particular, deciding whether a learned automaton is equivalent to the SUL is, in general, complex. Angluin already explained in [Ang87] that equivalence queries can be *approximated* with finitely many membership queries but it still requires the teacher to be able to decide whether an execution is valid. However, this assumes that the executions of the SUL follow some probability distribution, and that it is possible to obtain executions according to that distribution.

Since that approach is probabilistic, other methods exist that can decide the equivalence exactly, such as the so-called W-method [Cho78], at the cost of having to guess an upper bound over the number of states of the automaton that can represent the SUL.

[Cho78]: Chow (1978), "Testing Software Design Modeled by Finite-State Machines"

Learning with an inexperienced teacher. Some authors proposed variants of L^* where the teacher is assumed to be *inexperienced*: it may sometimes answer "I do not know" instead of a yes or no answer to membership queries. This means that the learner now has to play with incomplete information. See, for instance, [LN12] for a proper definition of inexperienced teacher

[LN12]: Leucker et al. (2012), "Learning Minimal Deterministic Automata from Inexperienced Teachers" and [Khm+22] for a study of the robustness of L^* when there is (random) noise in the communications between the learner and the teacher, *i.e.*, when the answers of the learner may be altered by some external factors.

[Khm+22]: Khmelnitsky et al. (2022), "Analyzing Robustness of Angluin's L* Algorithm in Presence of Noise"

Learning with information about the SUL. In order to learn complex systems, it may be required to give the teacher some information about the SUL, *i.e.*, to see it as a "gray box" (we partially know the SUL) or as a "white box" (we know exactly the SUL). This may also help improve the complexity of a learning algorithm, usually given by the number of queries asked by the learner. For instance, in [Ber+21], the authors assume they have information about the SUL in the form of an over-approximation of it. Similarly, in [AR16], the authors assume the SUL can be decomposed into two smaller systems, one of which they know in advance. In [MO20], the teacher is assumed to already have an executable automaton representation of SUL. This helps them learn the automaton directly and to do so more efficiently than other active learning algorithms for the same task. Finally, in [Gar+20] it is assumed that constraints satisfied along the run of a system can be made visible. They leverage this (tainting technique) to give a scalable learning algorithm for register automata.

[Ber+21]: Berthon et al. (2021), "Active Learning of Sequential Transducers with Side Information About the Domain"

[AR16]: Abel et al. (2016), "Gray-Box Learning of Serial Compositions of Mealy Machines"

[MO20]: Michaliszyn et al. (2020), "Learning Deterministic Automata on Infinite Words"

[Gar+20]: Garhewal et al. (2020), "Grey-Box Learning of Register Automata"

Learning automata with resources. As introduced above, it may be useful or required to allow automata to use *resources*, which permits the learning of more complex systems. There are many various possibilities and we list only a few of them here.

Counter When an automaton has to count something that has no upper bound, it cannot do so simply with its (finite) set of states. Hence, there exist families of automata with a single natural *counter*, which can be incremented or decremented along the executions of the automaton. Moreover, the behavior (*i.e.*, which edge of the graph to follow when reading a symbol) may depend on the counter value. See [NL10; FR95]. These automata will be the focus of Part II.

Stack As simply counting does not always suffice (see the above example on algebraic expressions), we may extend an automaton with a *stack*. We call such automata *pushdown automata*. Isberner provided a learning algorithm for a subfamily of pushdown automata in his PhD thesis [Isb15]. Part III presents this subfamily with more details.

Registers While stacks can be used to model various systems, the allowed operations on the stack are limited. For instance, only the top of the stack can be observed. If one needs more expressivity than what is possible by using counters or stacks, *registers*, which are a finite number of "memory cells" that can hold any value, can be considered. See [Gar+20; IHS14a] for learning algorithms, for instance.

Clocks Some systems must satisfy timing constraints. One possible way to model them is to consider *timed automata* which use *clocks* (real valued variables that measure the elapsed time). Learning algorithms for various restrictions over the clocks exist [Tan+22; Wag23; An+20; TZA24]. Part IV provides a new learning algorithm for a subfamily of timed automata.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

[FR95]: Fahmy et al. (1995), "Efficient Learning of Real Time One-Counter Automata"

[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

[Gar+20]: Garhewal et al. (2020), "Grey-Box Learning of Register Automata"

[IHS14a]: Isberner et al. (2014), "Learning register automata: from languages to program structures"

[Tan+22]: Tang et al. (2022), "Learning Deterministic One-Clock Timed Automata via Mutation Testing"

[Wag23]: Waga (2023), "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization"

[An+20]: An et al. (2020), "Learning One-Clock Timed Automata"

[TZA24]: Teng et al. (2024), "Learning Deterministic Multi-Clock Timed Automata"

Finally, there are also automata without resources but that encode different types of executions and that can be actively learned. For instance, it is possible to learn automata whose executions are of infinite length (*i.e.*, we assume the system runs forever) [AF16; MP95], or automata that have predicates over a domain of concrete symbols instead of said symbols [FFZ23; DD17].

[AF16]: Angluin et al. (2016), "Learning regular omega languages"
[MP95]: Maler et al. (1995), "On the Learnability of Infinitary Regular Sets"
[FFZ23]: Fisman et al. (2023), "Inferring Symbolic Automata"
[DD17]: Drews et al. (2017), "Learning Symbolic Automata"

Model checking. Finally, once a model is obtained, it is possible to apply model checking techniques to check whether the SUL behaves as expected, *e.g.*, [PVY02; GPY06]. For instance, once can check whether a given state, representing a catastrophic situation is always avoided. Model checking of course depends on the actual model that is learned and often requires to abstract it in order to ease the reasoning. We refer to [Cla+18; BK08] for introductions on model checking, in general.

[PVY02]: Peled et al. (2002), "Black Box Checking"
[GPY06]: Groce et al. (2006), "Adaptive Model Checking"
[Cla+18]: Clarke et al. (2018), *Handbook of Model Checking*
[BK08]: Baier et al. (2008), *Principles of model checking*

It is noteworthy that some authors provided algorithms that perform model checking *while* learning (see, *e.g.*, [Aic+18; San23]), or to correct erroneous models as in [GPY06], in the sense that the process starts with a model that does not behave exactly as the concrete system and refines it through learning, until either an error is identified in the system or the model correctly represents it.

[Aic+18]: Aichernig et al. (2018), "Model Learning and Model-Based Testing"
[San23]: Sankur (2023), "Timed Automata Verification and Synthesis via Finite Automata Learning"

1.3. Contributions

The contributions presented in this thesis lie in the fourth and fifth categories listed above. That is, we provide two learning algorithms for two families of automata extended with resources, and one validation algorithm that relies on learning an automaton which is then further abstracted. We here give a brief overview.

- Firstly, we provide a learning algorithm for a family of automata extended with a natural counter, called *realtime*[2] *one-counter automata* [BPS22]. The counter can be incremented or decremented along the transitions. Furthermore, it can be tested against zero, allowing different behaviors based on the current counter value.
  Since the counter does not have an upper bound in general, the number of pairs of a state and a counter value is potentially infinite, meaning that learning the behavior of a system requires special care. Inspired by [NL10], we provide a finite characterization of this behavior that can be learned by querying the system, and from which a one-counter automata can be extracted. We show that the algorithm constructs a number of hypotheses (before finding a "good" one) that is polynomial in the size of this characterization but requires exponentially many interactions to do so.
- Secondly, we focus on JSON documents, which can be used to store and transfer information in a way that is easily readable by a human and by a computer. We want to verify whether the document is valid with regards to a set of constraints, given as a JSON schema. The classical algorithm explores the constraints and the document in parallel. While this works well when the document is already fully known, it is not

2: *Realtime* means that the automaton is deterministic and every transition must read a symbol.

[BPS22]: Bruyère et al. (2022), "Learning Realtime One-Counter Automata"

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

optimal in a streaming context, *i. e.*, when the document is received piece by piece (which happens when a document is sent via a network, for instance). That is, in the worst case, the classical approach requires to retain the complete document in memory. This thus limits its usage in environments where the memory has to be optimized (such as a web server that must process many documents in parallel).
Our new approach [BPS23] first learns an automaton augmented with a stack (via the learning algorithm of [Isb15]) accepting a (strict) subset of the set of valid documents. This automaton is then used to efficiently decide whether a document is valid (*i. e.*, we are no longer restricted to the learned subset), without needing to store the whole document. That is, our validation algorithm has a lower overall memory requirement, at the cost of needing more time to validate a document, as observed on experimental results.

[BPS23]: Bruyère et al. (2023), "Validating Streaming JSON Documents with Learned VPAs"

[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

- Thirdly, we study automata whose resources are timers that can be used to encode timing constraints [Bru+23; Bru+24]. A timer is started at some value and decreases over time. Then, when it reaches zero, a special event occurs that must be handled, similarly to interruptions in a processor.
  It may happen that multiple timers reach zero at the same time, or that the user provides an input exactly at the same time a timer times out. In these cases, the model has a non-deterministic behavior as the automaton may decide to process these events in any order. We study the timed behavior of such an automaton and provide conditions ensuring that any untimed behavior can be observed without this non-determinism.
  Finally, we give an active learning algorithm, requiring a factorial number of interactions in the number of timers, and polynomial in the number of states. As, in practice, the number of timers remain relatively small, we claim that our algorithm can be used for real-world applications [Bru+24].

[Bru+23]: Bruyère et al. (2023), "Automata with Timers"

[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

1.4. Outline

High-level structure. Besides this introduction and the conclusion, the present document is split into four parts. Part I introduces general notations used throughout the whole document, and properly defines automata and Mealy machines (which are automata that can output symbols during their executions).

The remaining three parts each correspond to one of the items listed in the last section and start with a chapter that introduces the specific tools and problems. Then, our contributions are presented in one or two chapters. Technical details and proofs are deferred to the appendix of each part.

Part content. Part I, Preliminaries, introduces the notations used throughout the whole document, as well as automata and Mealy machines in Chapter 2. Furthermore, two learning algorithms are explained in Chapter 3: L^* for deterministic finite automata, and $L^\#$ for Mealy machines.

Part II, Learning Realtime One-Counter Automata, which is based on [BPS22], gives our active learning algorithm for realtime one-counter automata.

[BPS22]: Bruyère et al. (2022), "Learning Realtime One-Counter Automata"

- Chapter 4 first gives an overview of one-counter automata in general, draws a hierarchy of these families, and summarizes an active learning algorithm for a subfamily, called *visibly one-counter automata* [NL10].
- Chapter 5 then properly introduces realtime one-counter automata and explains, with examples, the key concepts of our learning algorithm, based on L^*.
- Appendix A contains the technical details and proofs of this part.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

Part III, Validating JSON Documents, based on [BPS23], gives our validation algorithm for JSON documents in a streaming context.

[BPS23]: Bruyère et al. (2023), "Validating Streaming JSON Documents with Learned VPAs"

- Chapter 6 first defines JSON documents and schemas (which are the way we express the constraints), introduces the abstractions and restrictions we consider to simplify the models and the classical validation algorithm used in many applications.
- In Chapter 7, we define *visibly pushdown automata* (the exact subfamily of pushdown automata we learn via the TTT learning algorithm [Isb15]), claim that any JSON schema (under our abstractions and restrictions) can be represented as a visibly pushdown automaton, and then give a new validation algorithm that does not require to hold the whole document in memory.
- Appendix B contains the technical details and proofs of this part.

Part IV, Mealy Machines with Timers, based on [Bru+23; Bru+24], focuses on automata with timers (rather, Mealy machines with timers), which form a subfamily of timed automata.

[Bru+23]: Bruyère et al. (2023), "Automata with Timers"
[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

- Chapter 8 first introduces timed Mealy machines (*i.e.*, automata with clocks that can produce outputs) and summarizes well-known key results and tools.
- Chapter 9 then properly defines Mealy machines with timers, proves that they form a subset of timed Mealy machines, before adapting some tools and results to fit within the timer context. Precisely, we adapt regions and zones, and show that deciding whether a state is reachable remains PSPACE-complete. Finally, we focus on the problem to decide whether every untimed run of a machine can be observed by some timed run in which all delays are non-zero, *i.e.*, such that we never have two events occurring at the same time.
- Chapter 10 gives, with examples, the key concepts of our active learning algorithm for Mealy machines with timers, based on $L^{\#}$. We also discuss experimental results obtained from our implementation.
- Appendix C contains the technical details and proofs of this part.

Finally, Part V, Conclusion, naturally ends this thesis by summarizing our contributions and emphasizing some prospects for the future.

Reading tips. The contribution parts are written to be readable as independently as possible. That is, Parts II to IV can all be read without having to read the preceding parts.[3] The only requirement is to first read Part I.

3: Definitions that are required in multiple parts are repeated.

At the end of the document, one can find an index, a list of notations, and a list of abbreviations.

The LaTeXclass used to typeset this document is available on GitHub[4]. This layout allows to print citations[5] and various other texts in the margins. We will use it to recall definitions, propositions, theorems, and so on, when seeking their first occurrence would necessitate going back multiple pages. Moreover, footnotes are replaced by "sidenotes".

4: `https://github.com/DocSkellington/ThesisLaTeXStyle`

5: Of course, a complete bibliography is available at the end of the document.

Funding. This thesis's author is a Research Fellow of the Fonds de la Recherche Scientifique – FNRS in the *ARTWORK* project, supervised by Véronique Bruyère (University of Mons) and Guillermo A. Pérez (University of Antwerp – Flanders Make) who is supported by the Belgian FWO "SAILor" project (G030020N). Our contributions presented in Part IV were partially funded by the NWO TOP project "GIRLS" (612.001.852).

Part I.

Preliminaries

Preliminaries 2.

The objective of the first part *Preliminaries* is to introduce the notions and notations that are used throughout this thesis. In particular, this chapter defines *automata* and *Mealy machines*, while Chapter 3 summarizes two learning algorithms that serve as the basis for the remaining parts.

Chapter contents

2.1. Mathematical Notations

We respectively denote by $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ the sets of natural numbers, integers, and real numbers. Furthermore, $\mathbb{N}^{>0}$ denotes the set of positive naturals, *i. e.*, $\mathbb{N}^{>0} = \{x \in \mathbb{N} \mid x > 0\}$. Similarly, $\mathbb{R}^{\geq 0} = \{x \in \mathbb{R} \mid x \geq 0\}$ and $\mathbb{R}^{>0} = \{x \in \mathbb{R} \mid x > 0\}$.

Let A be a set. We write $|A|$ for the cardinality of A. We say that a relation $\sim\ \subseteq A \times A$ is

- *reflexive* if, for all $a \in A$, $a \sim a$,
- *symmetric* if, for all $a, b \in A$, $a \sim b \Rightarrow b \sim a$, and
- *transitive* if, for all $a, b, c \in A$, $a \sim b \wedge b \sim c \Rightarrow a \sim c$.

If $\sim$ is reflexive, symmetric, and transitive, then it is called an *equivalence relation*. Whenever $\sim$ is an equivalence relation, we write $[\![a]\!]_\sim$ for the *equivalence class* of $a \in A$. The *index* of an equivalence relation is the cardinality of its set of equivalence classes.

We write $f : A \rightharpoonup B$ for a function f that maps some elements of A to some element of B. Then, the *domain* of f, denoted by $\mathsf{dom}(f)$, is the set of all elements a of A for which $f(a)$ is defined, while the *range* of f, denoted by $\mathsf{ran}(f)$, is the set of all elements b of B such that $f(a) = b$ for some $a \in \mathsf{dom}(f)$. When $\mathsf{dom}(f) = A$, we say that f is *total*. Otherwise, we say that it is *partial*. We write $f : A \to B$ to highlight that f is total. Finally, given two functions $f : A \rightharpoonup B$ and $g : B \rightharpoonup C$, the composition of g and f is the function $g \circ f : A \rightharpoonup C$ such that for all $x \in \mathsf{dom}(f)$ such that $f(x) \in \mathsf{dom}(g)$, we have $(g \circ f)(x) = g(f(x))$.

Finally, given a set A, we write $\mathbb{I}_A$ for the *identity relation* on A, *i. e.*, $\mathbb{I}_A = \{(a, a) \mid a \in A\}$.

2.2. Automata and Mealy machines

Let us introduce *automata* (that cannot use any resources). In short, an automaton is a state machine that processes sequences of input symbols, called (input) words, and returns a boolean indicating whether the word is “valid”.

An *alphabet* Σ is a non-empty finite set of *symbols*. A *word over* Σ is a finite sequence of symbols from Σ, and the *empty word* is denoted by ε. The set of all words over Σ is denoted by Σ^*.

The *concatenation* of two words $u, v \in \Sigma^*$ is denoted by $u \cdot v$. We sometimes forego the $\cdot$ to simply write uv.

A relation $\sim \subseteq \Sigma^* \times \Sigma^*$ is a *right-congruence*, or simply a *congruence*, if $a \cdot c$ and $b \cdot c$ are in relation whenever a and b are in relation, for every a, b, c in Σ^*. That is,

$$\forall a, b \in \Sigma^* : a \sim b \Rightarrow \forall c \in \Sigma^* : a \cdot c \sim b \cdot c.$$

A *language* L is a subset of Σ^*. Given a word $w \in \Sigma^*$ and a language $L \subseteq \Sigma^*$, the *set of prefixes of* w is

$$\mathit{Pref}(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, w = uv\}$$

and the *set of prefixes of* L is

$$\mathit{Pref}(L) = \bigcup_{w \in L} \mathit{Pref}(w).$$

Similarly, we have the sets of *suffixes*

$$\mathit{Suff}(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, w = vu\}$$

and

$$\mathit{Suff}(L) = \bigcup_{w \in L} \mathit{Suff}(w).$$

Moreover, L is said to be *prefix-closed* (resp. *suffix-closed*) if $L = \mathit{Pref}(L)$ (resp. $L = \mathit{Suff}(L)$).

2.2.1. Automata

An automaton is a (potentially infinite) state machine that reads a word over some alphabet Σ and either *accepts* or *rejects* the word. We first define nondeterministic automata where, for each state and symbol, there are multiple possible target states. As we explain below when we define the semantics, the actual successor state is arbitrarily selected among them.

Definition 2.2.1 (Automaton). An *automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where:

- ▶ Σ is an alphabet,
- ▶ Q is a non-empty set of *states*, with $q_0 \in Q$ the *initial state*,
- ▶ $F \subseteq Q$ is a set of *final states*, and
- ▶ $\delta : (Q \times \Sigma) \times Q$ is a *transition relation*. We write $\delta(q, a)$ to denote the

set of states p such that $((q,a),p) \in \delta$, and $q \xrightarrow{a} p$ when $p \in \delta(q,a)$.

When $\delta(q,a)$ is defined for every state q and symbol a, $\mathcal{A}$ is *complete*.
An automaton $\mathcal{A}$ is called *deterministic* whenever $|\delta(q,a)| \leq 1$ for every q and a. In this case, δ can be seen as a (partial) *transition function*, *i.e.*,

$$\delta : Q \times \Sigma \rightharpoonup Q.$$

When Q is finite, we say that $\mathcal{A}$ is a *nondeterministic finite automaton* (*NFA*, for short). Finally, if Q is finite and $\mathcal{A}$ is deterministic, we say that $\mathcal{A}$ is a *deterministic finite automaton* (*DFA*, for short). An example of a DFA is given below.

When needed, we add a superscript to indicate which automaton is considered, *e.g.*, $Q^{\mathcal{A}}, q_0^{\mathcal{A}}$, *etc.* Missing symbols in $q \xrightarrow{a} p$ are quantified existentially, *e.g.*, $q \xrightarrow{a}$ means that there exists p such that $q \xrightarrow{a} p$. We highlight that there exists at most one such p when $\mathcal{A}$ is deterministic.

Semantics

Let us now define the semantics of an automaton, *i.e.*, how to decide whether a word belongs to the language the automaton is meant to encode. To do so, we introduce *runs*, which are the successions of transitions that are triggered while reading a word.

Definition 2.2.2 (Run)**.** A *run* of an automaton $\mathcal{A}$ either consists of a single state p_0 or a nonempty sequence of transitions

$$\pi = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} p_n.$$

We denote by *runs*$(\mathcal{A})$ the set of runs of $\mathcal{A}$.
We lift to words $a_1 \cdots a_n$ as usual: $p_0 \xrightarrow{a_1 \cdots a_n} p_n \in$ *runs*$(\mathcal{A})$ if there exists a run $p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} p_n \in$ *runs*$(\mathcal{A})$.

To highlight that $\delta(q,i)$ is not empty, we often write $q \xrightarrow{i}\ \in$ *runs*$(\mathcal{A})$. Note that when $\mathcal{A}$ is nondeterministic, there may be multiple runs from a given state p_0 and word w. Contrarily, the run π is uniquely determined if $\mathcal{A}$ is deterministic.

Defining when a word w is *accepted* by $\mathcal{A}$ is now easy: we simply check whether there exists a run $q_0 \xrightarrow{w} p$ with $p \in F$.

Definition 2.2.3 (Language of $\mathcal{A}$)**.** A run $\pi = q_0 \xrightarrow{w} p$ is said to be *accepting* if $p \in F$. Then, w is *accepted by* $\mathcal{A}$ if and only if there exists an accepting run reading w.
The *language of* $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of words labeling accepting runs, *i.e.*,

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists p \in F : q_0 \xrightarrow{w} p \in \textit{runs}(\mathcal{A})\}.$$

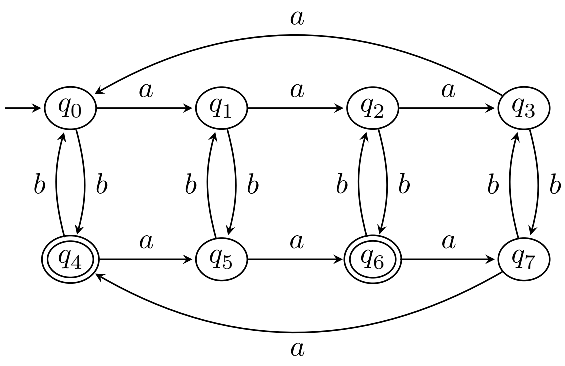


Figure 2.1: A sample DFA whose language is comprised of the words with an even number of a and an odd number of b.

Example 2.2.4. A 8-state DFA $\mathcal{A}$ over $\Sigma = \{a, b\}$ is given in Figure 2.1. The initial state q_0 is marked by a small arrow and the final states q_4 and q_6 are double-circled. The transitions between the states give the input symbol. A sample run of $\mathcal{A}$ is

$$\pi = q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_5 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_6.$$

As π starts in the initial state and ends in a final state, it is accepting, *i.e.*, $abbab$ is accepted by $\mathcal{A}$. However, $abba$ is not accepted by $\mathcal{A}$ as the unique run of $\mathcal{A}$ ends in the state q_2 which is not final:

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_5 \xrightarrow{b} q_1 \xrightarrow{a} q_2.$$

One can show that the language of $\mathcal{A}$ is comprised of all words with an even number of a and an odd number of b.

We highlight that any deterministic automaton is also a nondeterministic automaton. For finite automata, it is noteworthy that any NFA can be converted into a DFA [HU79]. That is, any language that is accepted by a DFA can be accepted by an NFA, and vice-versa. In other words, the set of all languages accepted by DFAs is exactly the set of all languages accepted by NFAs.

[HU79]: Hopcroft et al. (1979), *Introduction to Automata Theory, Languages and Computation*

Definition 2.2.5 (Regular language)**.** A language L is called *regular* if there exists a DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

The set of all regular languages is closed under complementation, union, and intersection.

We say that a finite automaton is *minimal* if for any DFA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$, we have $|Q^{\mathcal{A}}| \leq |Q^{\mathcal{B}}|$. For regular languages, this minimal automaton is unique and can be computed from a congruence over the words, called the *Myhill-Nerode congruence*. We here define this equivalence relation for any language. In short, it groups together words that have the same behaviors when extended. We then state that a language is regular if and only if the relation has finitely many classes.

Definition 2.2.6 (Myhill-Nerode congruence)**.** Let $L \subseteq \Sigma^*$ be a language. Two words $u, v \in \Sigma^*$ are said to be *L-equivalent*, denoted by $u \sim_L v$, if

$$\forall w \in \Sigma^* : uw \in L \Leftrightarrow vw \in L.$$

Observe that $\sim_L$ is a congruence, by definition.

Theorem 2.2.7 ([HU79])**.** *A language L is regular if and only if the index of $\sim_L$ is finite.*

Example 2.2.8. Let L be the language comprised of words with an even number of a and an odd number of b. Then, $\varepsilon \nsim_L a$ as $\varepsilon \cdot b = b \in L$ but $a \cdot b \notin L$. Contrarily, $\varepsilon \sim_L aa$.
There are four equivalence classes for $\sim_L$:

- all the words with an even number of a and an even number of b,
- all the words with an even number of a and an odd number of b,
- all the words with an odd number of a and an even number of b, and
- all the words with an odd number of a and an odd number of b.

Since there are finitely many classes, L is regular. Recall that Figure 2.1 gives a DFA accepting L.

Let us now define a deterministic automaton $\mathcal{A}$ from $\sim_L$ such that $\mathcal{L}(\mathcal{A}) = L$. In short, the set of states of $\mathcal{A}$ is exactly the set of equivalence classes of $\sim_L$ and the transitions follow naturally the classes: we go from $[\![w]\!]_{\sim_L}$ to $[\![wa]\!]_{\sim_L}$.[1]

1: Recall that $\sim_L$ is a congruence. Hence, this approach makes sense.

Definition 2.2.9 (Automaton from $\sim_L$)**.** Let $L \subseteq \Sigma^*$ be a language. We define the automaton $\mathcal{A}_{\sim_L} = (\Sigma, Q, q_0, F, \delta)$ with

- $Q = \{[\![w]\!]_{\sim_L} \mid w \in \Sigma^*\}$,
- $q_0 = [\![\varepsilon]\!]_{\sim_L}$,
- $F = \{[\![w]\!]_{\sim_L} \mid w \in L\}$, and
- $\delta : Q \times \Sigma \to Q$ is the (total) function defined such that for all $w \in \Sigma^*$ and $a \in \Sigma$,
$$[\![w]\!]_{\sim_L} \xrightarrow{a} [\![w \cdot a]\!]_{\sim_L} \in \mathit{runs}(\mathcal{A}_{\sim_L}).$$

Thanks to Theorem 2.2.7, we know that $\sim_L$ has finitely many equivalence classes when L is regular, meaning that $\mathcal{A}_{\sim_L}$ is a DFA. Furthermore, one can show that $\mathcal{A}_{\sim_L}$ is the minimal DFA accepting L [HU79]. Chapter 3 introduces a learning algorithm for DFAs, which, given a regular language L, identifies every equivalence class of $\sim_L$ and constructs the minimal DFA accepting L.

For instance, Figure 2.2 gives the minimal 4-state DFA constructed from the $\sim_L$ relation of Example 2.2.8.

2.2.2. Mealy machines

Let us now move towards *Mealy machines* which are also state machines. Unlike an automaton, a Mealy machine outputs a symbol at each transition,

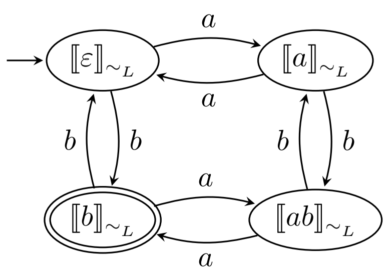


Figure 2.2: The minimal DFA accepting the same language as the DFA of Figure 2.1.

instead of a Boolean value at the end of a run. Hence, we require two alphabets: an input alphabet (denoted hereafter I) and an output alphabet (O). For simplicity, we only define deterministic Mealy machines.

Definition 2.2.10 (Mealy machine)**.** A Mealy machine *(*MM*, for short)* is a tuple $\mathcal{M} = (I, O, Q, q_0, \delta)$ where:

- ▶ I and O are the input and output alphabets,
- ▶ Q is a non-empty set of states, with $q_0 \in Q$ the initial state, and
- ▶ $\delta : Q \times I \rightharpoonup Q \times O$ is the transition function. We write $q \xrightarrow{i/o} p$ when $\delta(q, i) = (p, o)$.

We say that $\mathcal{M}$ is *complete* if $\delta(q, i)$ is defined for every state q and input symbol i.

Again, we add a superscript to indicate which MM is considered, *e. g.*, $Q^{\mathcal{M}}$, *etc.*. Missing symbols in $q \xrightarrow{i/o} p$ are quantified existentially, *e. g.*, $q \xrightarrow{i}$ means that there exist $p \in Q$ and $o \in O$ such that $q \xrightarrow{i/o} p$.

Semantics

While an MM does not have a notion of accepted language, it produces an output word (*i. e.*, a word over O) for each input word (*i. e.*, over I) that can be read. That is, an MM can be seen as a function from I^* to O^*. The adaptation of runs to MMs is straightforward.

Definition 2.2.11 (Run)**.** A *run* of an MM $\mathcal{M}$ either consists of a single state p_0 or a nonempty sequence of transitions

$$\pi = p_0 \xrightarrow{i_1/o_1} p_1 \xrightarrow{i_2/o_2} \cdots \xrightarrow{i_n/o_n} p_n.$$

We denote by *runs*$(\mathcal{M})$ the set of runs of $\mathcal{M}$.
We lift to words $i_1 \cdots i_n$ as usual: $p_0 \xrightarrow{i_1 \cdots i_n} p_n \in \mathit{runs}(\mathcal{M})$ if there exists a run $p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_n} p_n \in \mathit{runs}(\mathcal{M})$.
For an input word w, we write $\mathit{output}^{\mathcal{M}}(w)$ for the output word obtained by concatenating the output symbols of the run $q_0 \xrightarrow{w}$.

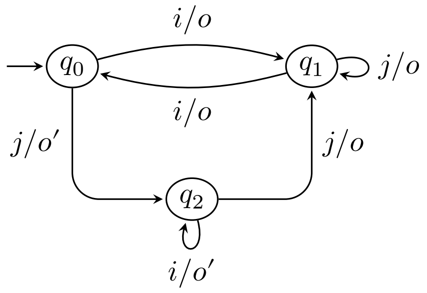


Figure 2.3: A sample MM.

Again, to highlight that $\delta(q, i)$ is defined, we often write $q \xrightarrow{i} \in$ *runs*$(\mathcal{M})$. Note that any run π is uniquely determined by its first state and word, as $\mathcal{M}$ is deterministic. Hence, for every word w, there exists a most one run $q_0 \xrightarrow{w}$, meaning that there exists at most one output word, *i. e.*, *output*$^{\mathcal{M}}(w)$ returns a word (if the run exists).

Example 2.2.12. A 4-state MM $\mathcal{M}$ is given in Figure 2.3, with $I = \{i, j\}$ and $O = \{o, o'\}$. Transitions show the input and the output, separated by a slash. A sample run is

$$\pi = q_0 \xrightarrow{j/o'} q_2 \xrightarrow{i/o'} q_2 \xrightarrow{j/o} q_1 \xrightarrow{j/o} q_1.$$

Finally, we conclude by defining when two MMs are deemed equivalent: for every input word w, both MMs output the same word.

Definition 2.2.13 (Equivalence of MMs)**.** Two MMs $\mathcal{M}$ and $\mathcal{N}$ over I and O are *equivalent*, denoted by $\mathcal{M} \approx \mathcal{N}$, if for all $w \in I^*$, *output*$^{\mathcal{M}}(w) =$ *output*$^{\mathcal{N}}(w)$.

Active Learning of DFAs and MMs | 3.

In this chapter, we introduce *automata learning algorithms* which are tools to automatically construct a DFA (or an MM). These algorithms can be split into two families:[1]

1: Chapter 1 gives a more general introduction to automata learning.

Passive learning algorithms Given a finite set $\mathcal{P}$ of words that are in the target language and a finite set $\mathcal{N}$ of words that are not in the target language, construct a DFA $\mathcal{A}$ that accepts at least all words of $\mathcal{P}$ and accepts none of the words of $\mathcal{N}$, *i.e.*, such that $\mathcal{P} \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}) \cap \mathcal{N} = \emptyset$. See, *e.g.*, [BF72; HV10; NJ13].

Active learning algorithms Given a target language L, build a DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$, by querying L, as introduced by Dana Angluin [Ang87].

[BF72]: Biermann et al. (1972), "On the Synthesis of Finite-State Machines from Samples of Their Behavior"

[HV10]: Heule et al. (2010), "Exact DFA Identification Using SAT Solvers"

[NJ13]: Neider et al. (2013), "Regular Model Checking Using Solver Technologies and Automata Learning"

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

In this thesis, we focus on active learning and provide here the main ideas behind two algorithms that serve as basis for the algorithms developed in Parts II and IV. We refer the reader to [Nei14] for a more complete introduction to learning automata (including passive algorithms).

[Nei14]: Neider (2014), "Applications of automata learning in verification and synthesis"

Chapter contents

3.1. Angluin's framework

Let us fix a regular language $L \subseteq \Sigma^*$. Our goal is to automatically learn a DFA accepting L. Active learning algorithms rely on the so-called *Angluin's framework*, which consists of two agents [Ang87]:

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

- The *teacher* who knows the target language L and can decide wether a given word belongs to L, and whether a DFA accepts L, and
- the *learner* who initially does not know anything about L but can interact with the teacher to gather knowledge and eventually construct a hypothesis DFA.

The learner can interact with the teacher through *queries*. A key point of Angluin's framework is that the teacher and the queries must be *minimally adequate*, in the sense that the teacher does not offer too much help. When learning regular languages, we authorize two types of queries: one to know whether a specific word belongs to L, and one to know whether a DFA accepts

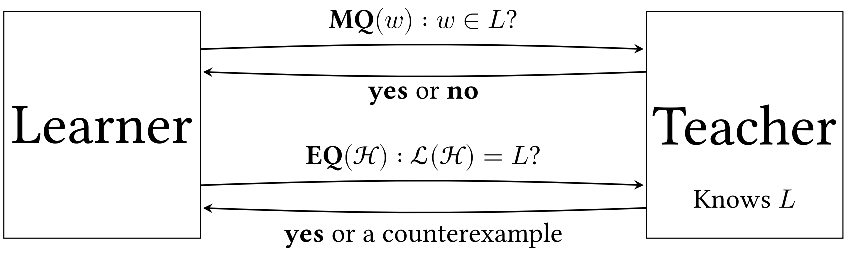

Learner
Teacher
Knows L
MQ$(w) : w \in L?$
yes or **no**
EQ$(\mathcal{H}) : \mathcal{L}(\mathcal{H}) = L?$
yes or a counterexample

Figure 3.1: Illustration of the Angluin's framework for DFAs.

L. Notice that, in a pure theoretical setting, the learner can eventually find a DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$ using only equivalence queries: simply enumerate every possible DFA until finding the good one. This is, of course, impractical and the reason why we need membership queries.

Definition 3.1.1 (Queries for DFAs)**.** Let L be the regular language of the teacher. A learner for DFAs can use two types of *queries*:

- A *membership query*, denoted by **MQ**(w), with w a word over Σ^*, returns whether $w \in L$.
- An *equivalence query*, denoted by **EQ**$(\mathcal{H})$, with $\mathcal{H}$ a DFA, returns
  - **yes** if $\mathcal{L}(\mathcal{H}) = L$, and
  - a word w such that $w \in \mathcal{L}(\mathcal{H}) \setminus L$ or $w \in L \setminus \mathcal{L}(\mathcal{H})$. Such a word w is called a *counterexample*.

Figure 3.1 gives a visual representation of the framework for DFAs. Every learning algorithm we consider in this thesis relies on Angluin's framework, or an extension of it that adds further queries. For instance, Parts II and IV will both add new queries, while the adaptation for MMs is given in Section 3.3.

In practice, it may be that equivalence cannot be directly implemented (if the system we want to learn is an actual black-box). In that case, as Angluin showed [Ang87], one can sample the system by performing many membership queries over random words. While this is not discussed in this thesis, [LY96] offers an overview of many sampling methods.

[LY96]: Lee et al. (1996), "Principles and methods of testing finite state machines-a survey"

The following two sections give the main ideas for two different active learning algorithms: L^* for DFAs [Ang87], and $L^\#$ for MMs [Vaa+22]. In particular, while we give interesting theorems and claims, we do not provide their proofs here. We refer the reader to the corresponding papers for more details.

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

3.2. L^*

Assume that the target language $L \subseteq \Sigma^*$ is regular. In short, the L^* algorithm we introduce here identifies every equivalence class of the Myhill-Nerode congruence $\sim_L$ (see Definition 2.2.6) by using **MQ** and **EQ** and storing the gathered knowledge inside a table.

Definition 2.2.6. For a language $L \subseteq \Sigma^*$ and two words $u, v \in \Sigma^*$, we write $u \sim_L v$ if for all $w \in \Sigma^*$, we have $uw \in L \Leftrightarrow vw \in L$.

This idea relies on the fact that one can define an infinite table whose rows and columns are words over Σ: for each row u and column v, the cell $u \cdot v$

stores whether $u \cdot v \in L$. Then, one can group together the rows that have exactly the same contents, *i. e.*, two rows u and v are equivalent if and only if $uw \in L \Leftrightarrow vw \in L$ for every column w. Notice that this exactly the definition of $\sim_L$. If L is regular, we know that there are finitely many equivalence classes for $\sim_L$, by Theorem 2.2.7, *i. e.*, finitely many groups of rows. The L^* algorithm identifies a finite number of rows and columns that are sufficient to represent and distinguish every class of $\sim_L$.

Theorem 2.2.7. A language L is regular if and only if the index of $\sim_L$ is finite.

We first define the table that is used to store the observations, and how to construct a DFA hypothesis from it. Then, we provide the main loop of L^* and how to process a counterexample in Section 3.2.2. Finally, Section 3.2.3 gives an example of a complete execution of L^*. Before doing so, let us state that the algorithm eventually terminates and its complexity.

Theorem 3.2.1 ([Ang87]). *Let n be the size of the minimal DFA accepting the target language L, and ζ be the length of the longest counterexample provided by the teacher. Then,*

- *the L^* algorithm eventually terminates and returns a DFA accepting L,*
- *and uses at most n equivalence queries and $\mathcal{O}\left(\zeta n^2 |\Sigma|\right)$ membership queries.*

Before delving into the details, we highlight that there are many variations of L^* that all focus on identifying the equivalence classes of $\sim_L$ but use different data structures to do so. For instance, Kearns and Vazirani [KV94] use an observation *tree* instead of a table, while Isberner *et al.* [IHS14b] rely on three different trees to store the results obtained from the queries and refine the hypothesis in an efficient way. While the validation algorithm that will be introduced in Part III relies on the algorithm from [IHS14b], it will do so by using it as a black-box. Hence, we only introduce L^*, which is the basis of our learning algorithm in Part II. Section 3.3 introduces another learning algorithm, called $L^{\#}$, which does not work by identifying the equivalence classes of $\sim_L$.

[KV94]: Kearns et al. (1994), *An Introduction to Computational Learning Theory*

[IHS14b]: Isberner et al. (2014), "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning"

3.2.1. Observation table

The main data structure of L^* is called the *observation table*, denoted by $\mathcal{O}$, which is a finite sub-table of the infinite table. That is, we have a finite set of words that label each row, and a finite number of words labelling each column. We then define an equivalence relation $\equiv_{\mathcal{O}}$ that groups together words labeling rows with exactly the same contents. We refine this table until some properties are satisfied, allowing us to construct a DFA hypothesis from the equivalence classes of $\equiv_{\mathcal{O}}$.

Definition 3.2.2 (Observation table [Ang87]). An *observation table* is a tuple $\mathcal{O} = (R, S, T)$ where:

- $R \subsetneq \Sigma^*$ is a finite prefix-closed set of *representatives*,
- $S \subsetneq \Sigma^*$ is a finite suffix-closed set of *separators*,
- $T : (R \cup R\Sigma) \cdot S \to \{\mathbf{no}, \mathbf{yes}\}$ is a (total) function that stores the

	ε	b
ε	**no**	**yes**
a	**no**	**no**
b	**yes**	**no**
ab	**no**	**no**
aba	**yes**	**no**
aa	**no**	**yes**
ba	**no**	**no**
bb	**no**	**yes**
abb	**no**	**no**
$abaa$	**no**	**no**
$abab$	**no**	**yes**

Figure 3.2: A sample observation table.

gathered knowledge on L:

$$\forall w \in (R \cup R\Sigma) \cdot S : T(w) = \begin{cases} \textbf{yes} & \text{if } w \in L \\ \textbf{no} & \text{otherwise.} \end{cases}$$

Filling the table is easy: for every (extended) representative u and separator v such that $T(u \cdot v)$ is not yet known, ask **MQ**$(u \cdot v)$ and set $T(u \cdot v)$ accordingly.

Notice that T also holds information about $R\Sigma$ (*i. e.*, representatives extended with a symbol). This will be useful when constructing the hypothesis to know how to define the transitions that leave the state corresponding to a representative. We first define the equivalence class the table encodes, followed by the constraints $\mathcal{O}$ must satisfy for a hypothesis to be constructed, and, finally, the hypothesis construction itself.

Equivalence relation from $\mathcal{O}$

As the Myhill-Nerode congruence groups together two words u and v such that $u \cdot w \in L \Leftrightarrow v \cdot w \in L$ for every possible w, we define an equivalence relation from $\mathcal{O}$ that follows the same idea, except that u and v must be rows and w a column of the table.

Definition 3.2.3 (Equivalence relation of an observation table)**.** Let $\mathcal{O}$ be an observation table. For two words $u, v \in R \cup R\Sigma$, we write $u \equiv_{\mathcal{O}} v$ if and only if $T(u \cdot w) = T(v \cdot w)$ for all $w \in S$.

Example 3.2.4. Let L be the language of all words with an even number of a and an odd number of b. Figure 3.2 gives an example of an observation table $\mathcal{O}$ with $R = \{\varepsilon, a, b, ab, aba\}$ and $S = \{\varepsilon, b\}$. Then, the cell $b \cdot b$ contains **no**, as $b \cdot b \notin L$, while $T(aba \cdot \varepsilon) =$ **yes**.
We have that $\varepsilon \equiv_{\mathcal{O}} aa$ but $a \not\equiv_{\mathcal{O}} b$ as $T(a \cdot \varepsilon) =$ **no** $\neq T(b \cdot \varepsilon) =$ **yes**. It is

not hard to see that we have three equivalence classes:

$$\begin{aligned}
[\![\varepsilon]\!]_{\equiv_{\mathcal{O}}} &= \{\varepsilon, aa, bb, abab\}\\
[\![a]\!]_{\equiv_{\mathcal{O}}} &= \{a, ab, ba, abb, abaa\}\\
[\![b]\!]_{\equiv_{\mathcal{O}}} &= \{b, aba\}.
\end{aligned}$$

Notice that $\equiv_{\mathcal{O}}$ is coarser than $\sim_L$. The goal of the learner in the L^* algorithm is to ask membership and equivalence queries to refine $\equiv_{\mathcal{O}}$ until it coincides with $\sim_L$.

Closed and Σ-consistent table

In order to be able to construct a DFA hypothesis, we have to ensure that every row in $R\Sigma$ has an equivalent row in R (*i. e.*, every extension of a representative still has a known representative), and that $\equiv_{\mathcal{O}}$ is a valid congruence (*i. e.*, whenever $u \equiv_{\mathcal{O}} v$, it must hold that $ua \equiv_{\mathcal{O}} va$ for each $a \in \Sigma$).

Definition 3.2.5 (Closed and Σ-consistent table)**.** We say that an observation table $\mathcal{O}$ is

- *closed* if for all $u \in R\Sigma$, there exists $v \in R$ such that $u \equiv_{\mathcal{O}} v$, and *open* otherwise, and
- Σ-*consistent* if for all $u, v \in R$ and $a \in \Sigma$ such that $u \equiv_{\mathcal{O}} v$, it holds that $u \cdot a \equiv_{\mathcal{O}} v \cdot a$, and Σ-*inconsistent* otherwise.

When an observation table $\mathcal{O}$ is open, *i. e.*, when

$$\exists u \in R\Sigma, \forall v \in R : u \not\equiv_{\mathcal{O}} v,$$

it means that u must represent an equivalence class that is currently missing. We add u to R, which implies that new cells have to be filled, due to the new extended representatives ua, with $a \in \Sigma$, *i. e.*, we ask a membership query for each new cell.

When an observation table $\mathcal{O}$ is Σ-inconsistent, *i. e.*, when

$$\exists u \neq v \in R, a \in \Sigma : u \equiv_{\mathcal{O}} v \wedge u \cdot a \not\equiv_{\mathcal{O}} v \cdot a,$$

i. e., $T(u \cdot a \cdot w) \neq T(v \cdot a \cdot w)$ for some $w \in S$, it means that $a \cdot w$ allows us to distinguish the equivalence classes $[\![u]\!]_{\sim_L}$ and $[\![v]\!]_{\sim_L}$ and, thus, u and v should not belong in the same class of $\equiv_{\mathcal{O}}$. Hence, we add $a \cdot w$ as a new separator, and fill the new cells by asking membership queries.

Example 3.2.6. Let $\mathcal{O}$ be the observation table on the left of Figure 3.3. Notice that $b \not\equiv_{\mathcal{O}} \varepsilon$, as $T(b) =$ **yes** $\neq T(\varepsilon) =$ **no**. Hence, $\mathcal{O}$ is open and we add b to the representatives. Figure 3.3 gives the resulting observation table on the right.
Now, let us consider the observation table $\mathcal{O}'$ of Figure 3.2 (which we repeat in the margin). We have $a \equiv_{\mathcal{O}'} ab$ but $aa \not\equiv_{\mathcal{O}'} aba$, meaning that $\mathcal{O}'$ is Σ-inconsistent. More precisely, we have $T(a \cdot a \cdot \varepsilon) =$ **no** while $T(ab \cdot a \cdot \varepsilon) =$ **yes**.

	ε	b
ε	**no**	**yes**
a	**no**	**no**
b	**yes**	**no**
ab	**no**	**no**
aba	**yes**	**no**
aa	**no**	**yes**
ba	**no**	**no**
bb	**no**	**yes**
abb	**no**	**no**
$abaa$	**no**	**no**
$abab$	**no**	**yes**

	ε
ε	**no**
a	**no**
b	**yes**

	ε
ε	**no**
b	**yes**
a	**no**
ba	**no**
bb	**no**

Figure 3.3: On the left, an observation table that is open. On the right, the table obtained by resolving the openness.

	ε	b	a
ε	**no**	**yes**	**no**
a	**no**	**no**	**no**
b	**yes**	**no**	**no**
ab	**no**	**no**	**yes**
aba	**yes**	**no**	**no**
aa	**no**	**yes**	**no**
ba	**no**	**no**	**yes**
bb	**no**	**yes**	**no**
abb	**no**	**no**	**no**
$abaa$	**no**	**no**	**yes**
$abab$	**no**	**yes**	**no**

Figure 3.4: The observation table obtained by resolving the Σ-inconsistency of the table from Figure 3.2.

We thus add $a \cdot \varepsilon = a$ to the separators. Figure 3.4 gives the resulting table.

Hypothesis construction

Once $\mathcal{O}$ is closed and Σ-consistent, one can show that $\equiv_{\mathcal{O}}$ is a right-congruence. That means that a hypothesis (denoted by $\mathcal{H}_{\mathcal{O}}$) can be constructed using the same ideas as the construction of an automaton from the Myhill-Nerode congruence (see Definition 2.2.9), except that we consider the classes of $\equiv_{\mathcal{O}}$. As the index of $\equiv_{\mathcal{O}}$ is finite, the resulting automaton is a DFA.

Definition 3.2.7 (Automaton from $\equiv_{\mathcal{O}}$)**.** We construct the DFA $\mathcal{H}_{\mathcal{O}}$ with

- $Q^{\mathcal{H}_{\mathcal{O}}} = \{[\![w]\!]_{\equiv_{\mathcal{O}}} \mid w \in \Sigma^*\}$, with $q_0^{\mathcal{H}_{\mathcal{O}}} = [\![\varepsilon]\!]_{\equiv_{\mathcal{O}}}$,
- $F^{\mathcal{H}_{\mathcal{O}}} = \{[\![w]\!]_{\equiv_{\mathcal{O}}} \mid w \in L\}$, and
- $\delta^{\mathcal{H}_{\mathcal{O}}} : Q \times \Sigma \to Q$ is the (total) function defined such that for all $[\![w]\!]_{\equiv_{\mathcal{O}}}, [\![w \cdot a]\!]_{\equiv_{\mathcal{O}}} \in Q$ and $a \in \Sigma$,

$$[\![w]\!]_{\equiv_{\mathcal{O}}} \xrightarrow{a} [\![w \cdot a]\!]_{\equiv_{\mathcal{O}}} \in \mathit{runs}(\mathcal{H}_{\mathcal{O}}).$$

Example 3.2.8. Let $\mathcal{O}$ be the observation table of Figure 3.4. It is closed and Σ-consistent, allowing us to construct a hypothesis $\mathcal{H}_{\mathcal{O}}$ from it. We have

Definition 2.2.9. Let $L \subseteq \Sigma^*$ be a language. We define the automaton $\mathcal{A}_{\sim_L} = (\Sigma, Q, q_0, F, \delta)$ with

- $Q = \{[\![w]\!]_{\sim_L} \mid w \in \Sigma^*\}$,
- $q_0 = [\![\varepsilon]\!]_{\sim_L}$,
- $F = \{[\![w]\!]_{\sim_L} \mid w \in L\}$, and
- $\delta : Q \times \Sigma \to Q$ is the (total) function defined such that for all $w \in \Sigma^*$ and $a \in \Sigma$,

$$[\![w]\!]_{\sim_L} \xrightarrow{a} [\![w \cdot a]\!]_{\sim_L} \in \mathit{runs}(\mathcal{A}_{\sim_L}).$$

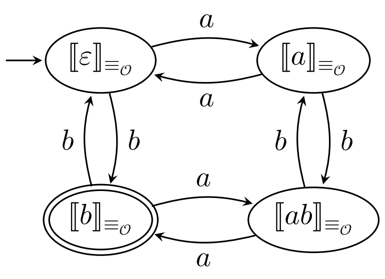

a
a
b
b
b
b
a
a
$[\![\varepsilon]\!]_{\equiv_{\mathcal{O}}}$
$[\![a]\!]_{\equiv_{\mathcal{O}}}$
$[\![b]\!]_{\equiv_{\mathcal{O}}}$
$[\![ab]\!]_{\equiv_{\mathcal{O}}}$

Figure 3.5: The hypothesis constructed from the closed and Σ-consistent table of Figure 3.4.

the following equivalence classes:

$$\begin{aligned}[\![\varepsilon]\!]_{\equiv_{\mathcal{O}}} &= \{\varepsilon, aa, bb, abab\},\\ [\![a]\!]_{\equiv_{\mathcal{O}}} &= \{a, abb\},\\ [\![b]\!]_{\equiv_{\mathcal{O}}} &= \{b, aba\},\end{aligned}$$

and

$$[\![ab]\!]_{\equiv_{\mathcal{O}}} = \{ab, ba, abaa\}.$$

One can verify that $\equiv_{\mathcal{O}}$ is a right-congruence. Figure 3.5 gives the resulting DFA.

3.2.2. Main loop

Algorithm 3.1 gives a pseudo-code for L^*. Initially, the observation table has ε as the unique representative and separator (meaning that $R\Sigma = \Sigma$) and is filled via **MQ**. Then, we make the table closed and Σ-consistent. We highlight that this may require many membership queries and iterations, as adding a new representative or separator may induce further refinements. While we do not provide it here, [Ang87] gives a proof that this always eventually terminates. Once $\mathcal{O}$ is closed and Σ-consistent, we construct a hypothesis $\mathcal{H}_{\mathcal{O}}$ and ask an equivalence query. If the answer is **yes**, we return $\mathcal{H}_{\mathcal{O}}$ (as we have identified a DFA accepting L). Otherwise, we have to process the provided counterexample.

Counterexample processing

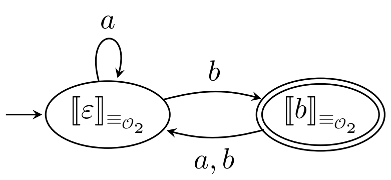
Let w be a counterexample returned by the teacher for some $\mathbf{EQ}(\mathcal{H}_{\mathcal{O}})$, *i.e.*, w is such that $w \in \mathcal{H}_{\mathcal{O}} \Leftrightarrow w \notin L$. Angluin [Ang87] proposes to handle w by adding all elements in $\mathit{Pref}(w)$ to R. That way, R remains prefix-closed and we are sure to have a representative for the missing equivalence class. It may be that the table is open or Σ-inconsistent after adding $\mathit{Pref}(w)$, requiring us to refine the table before constructing the next hypothesis.

Notice that the number of representatives grows with the length of w. So, if w is extremely long, we may add too many representatives, with regards to the equivalence classes of $\sim_L$ (*i.e.*, we may have many representatives for the same class). Although we only rely on adding all prefixes of w in this thesis, some works propose more efficient ways of handling w. For instance, Rivest

Algorithm 3.1: The L^* algorithm for DFAs [Ang87].

Require: The target language L
Ensure: A DFA accepting L is returned

```
1:  Initialize $\mathcal{O}$ with $R = S = \{\varepsilon\}$
2:  Fill $\mathcal{O}$ by asking **MQ**
3:  **while** true **do**
4:      Make $\mathcal{O}$ closed and $\Sigma$-consistent
5:      Construct the DFA $\mathcal{H}_{\mathcal{O}}$ from $\mathcal{O}$
6:      $w \leftarrow$ **EQ**$(\mathcal{H}_{\mathcal{O}})$
7:      **if** $w =$ **yes then**
8:          **return** $\mathcal{H}_{\mathcal{O}}$
9:      **else**
10:         ProcessCounterExample$(w)$
```



a
b
$[\![\varepsilon]\!]_{\equiv_{\mathcal{O}_2}}$
$[\![b]\!]_{\equiv_{\mathcal{O}_2}}$
a, b

Figure 3.6: The hypothesis constructed from $\mathcal{O}_2$.

and Schapire [RS93] perform a binary search to identify the shortest prefix of w that is still a counterexample. Furthermore, they show that adding a single suffix (obtained from the binary search) is sufficient to learn a new equivalence class.[2] This idea is also applied in the adaptation of L^* to MMs [SG09].

[RS93]: Rivest et al. (1993), "Inference of Finite Automata Using Homing Sequences"

2: In [RS93], S is not necessarily suffix-closed.

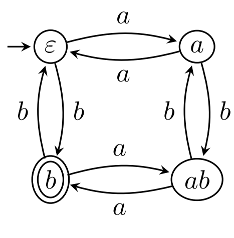
[SG09]: Shahbaz et al. (2009), "Inferring Mealy Machines"

For more details on various counterexample processing algorithms, we refer the reader to Isberner and Steffen's work [IS14]. The authors also introduce an *exponential* search (instead of binary) that is more efficient in practice, as it lowers the total number of input symbols used in membership queries while processing w.

[IS14]: Isberner et al. (2014), "An Abstract Framework for Counterexample Analysis in Active Automata Learning"

3.2.3. Complete example

Finally, we give an example of a complete run of L^* using the language L of all words with an even number of a and an odd number of b. This language is accepted by the DFAs of Figures 2.1 and 2.2 (which is repeated in the margin, for convenience).



Throughout the example, we number each table, starting with $\mathcal{O}_1$ whose only representative and separator is ε, *i. e.*, $R_1 = S_1 = \{\varepsilon\}$ and is given on the left of Figure 3.3. As explained in Example 3.2.6, this table is open due to b. Hence, we move b to R and obtain the table $\mathcal{O}_2$ such that $R_2 = \{\varepsilon, b\}$ and $S_2 = S_1 = \{\varepsilon\}$, which is given on the right of Figure 3.3 and repeated in the margin. This table is closed and Σ-consistent. We can thus construct the hypothesis $\mathcal{H}_{\mathcal{O}_1}$, shown in Figure 3.6.

	ε
ε	**no**
b	**yes**
a	**no**
ba	**no**
bb	**no**

	ε
ε	**no**
a	**no**
b	**yes**
ab	**no**
aba	**yes**
aa	**no**
ba	**no**
bb	**no**
abb	**no**
$abaa$	**no**
$abab$	**no**

Figure 3.7: The observation table $\mathcal{O}_3$ obtained by adding the prefixes of the counterexample aba.

We ask **EQ**$(\mathcal{H}_{\mathcal{O}_2})$ and receive the counterexample aba. We add all words in *Pref* $(aba) = \{\varepsilon, a, ab, aba\}$ as new representatives, *i.e.*, $R_3 = \{\varepsilon, b, a, ab, aba\}$. Figure 3.7 gives the resulting table $\mathcal{O}_3$. Observe that it is not Σ-consistent, as $\varepsilon \equiv_{\mathcal{O}_3} a$ but $\varepsilon \cdot b \not\equiv_{\mathcal{O}_3} a \cdot b$, meaning that we add b as a new separator, *i.e.*, $S_4 = \{\varepsilon, b\}$. The table $\mathcal{O}_4$ is given in Figure 3.2. As studied in Example 3.2.6, $\mathcal{O}_4$ is still Σ-inconsistent and we add a to S_4.

Finally, the resulting observation table $\mathcal{O}_5$, with $S_5 = \{\varepsilon, a, b\}$, is shown in Figure 3.4 (repeated in the margin) and is closed and Σ-consistent, as mentioned in Example 3.2.8. Moreover, the hypothesis $\mathcal{H}_{\mathcal{O}_5}$ is given in Figure 3.5. This time, **EQ**$(\mathcal{H}_{\mathcal{O}_5})$ returns **yes**, *i.e.*, we have correctly learned an automaton that accepts L. Notice that $\mathcal{H}_{\mathcal{O}_5}$ is isomorphic to the minimal DFA accepting L.

	ε	b	a
ε	**no**	**yes**	**no**
a	**no**	**no**	**no**
b	**yes**	**no**	**no**
ab	**no**	**no**	**yes**
aba	**yes**	**no**	**no**
aa	**no**	**yes**	**no**
ba	**no**	**no**	**yes**
bb	**no**	**yes**	**no**
abb	**no**	**no**	**no**
$abaa$	**no**	**no**	**yes**
$abab$	**no**	**yes**	**no**

3.3. $L^{\#}$

In this section, we introduce $L^{\#}$ [Vaa+22], which is an active learning algorithm for MMs. Contrarily to L^*, $L^{\#}$ does not try to identify the equivalence classes of the Myhill-Nerode congruence (adapted to MMs). Instead, it relies on a notion of *apartness* that proves that two states must have different behaviors. Another main difference is that $L^{\#}$ does not maintain auxiliary data structures to construct a hypothesis, but operates directly on the tree built from the observations made via queries. See [SG09; SHM11] for learning algorithms for MMs based on adapting of L^*.

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

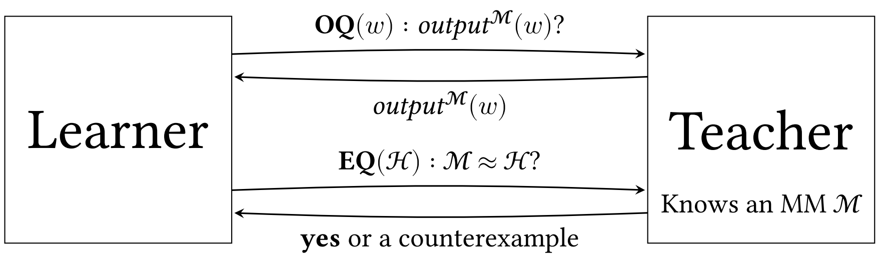
[SG09]: Shahbaz et al. (2009), "Inferring Mealy Machines"
[SHM11]: Steffen et al. (2011), "Introduction to Active Automata Learning from a Practical Perspective"

We first adapt Angluin's framework to MMs . This time, we assume the teacher knows a complete MM $\mathcal{M}$ and the goal of the learner is to construct an MM $\mathcal{H}$ such that $\mathcal{M} \approx \mathcal{H}$ (see Definition 2.2.13).

Definition 2.2.13. Two MMs $\mathcal{M}$ and $\mathcal{N}$ over I and O are *equivalent*, denoted by $\mathcal{M} \approx \mathcal{N}$, if for all $w \in I^*$, *output*$^{\mathcal{M}}(w) =$ *output*$^{\mathcal{N}}(w)$.

Definition 3.3.1 (Queries for MMs)**.** Let $\mathcal{M}$ be the MM of the teacher. A learner for MMs can use two types of *queries*:

- An *output query*, denoted by **OQ**(w), with w a word over I, returns *output*$^{\mathcal{M}}(w)$.
- An *equivalence query*, denoted by **EQ**$(\mathcal{H})$, with $\mathcal{H}$ an MM, returns
    - **yes** if $\mathcal{M} \approx \mathcal{H}$, and



Learner
Teacher
Knows an MM $\mathcal{M}$
$\mathbf{OQ}(w) : output^{\mathcal{M}}(w)?$
$output^{\mathcal{M}}(w)$
$\mathbf{EQ}(\mathcal{H}) : \mathcal{M} \approx \mathcal{H}?$
yes or a counterexample

Figure 3.8: Adaptation of Angluin's framework for MMs.

- a word w over I such that $output^{\mathcal{M}}(w) \neq output^{\mathcal{H}}(w)$ otherwise. Such a word w is called a *counterexample*.

Figure 3.8 gives a visual representation of the framework. Part IV will extend this framework to allow learning of MMs extended with *timers*. We highlight that the notations introduced here differ from those used in [Vaa+22], in order to ease Part IV.

The rest of this section is structured as follows. We start by defining the data structure of $L^{\#}$, give useful properties, and explain how to construct a hypothesis, assuming the data structure satisfies some constraints. This allows us to provide the main loop of $L^{\#}$ and how to process a counterexample in Section 3.3.2. Finally, a complete example of an execution of $L^{\#}$ is given in Section 3.3.3. Before that, let us state that the algorithm eventually terminates and its complexity.

Theorem 3.3.2 ([Vaa+22]). *Let n be the size of a minimal MM equivalent to the teacher's MM over the input alphabet I, and ζ be the length of the longest counterexample provided by the teacher. Then,*

- *the $L^{\#}$ algorithm eventually terminates,*
- *and uses at most $n-1$ equivalence queries and $\mathcal{O}\left(n^2|I| + n \log \zeta\right)$ membership queries.*

3.3.1. Observation tree

From an MM $\mathcal{M}$, we can construct an infinite Mealy machine equivalent to $\mathcal{M}$ by unfolding the runs of $\mathcal{M}$ into an infinite tree (of finite arity). Then, a "prefix-closed"[3] subset of states can be constructed, which we denote $\mathcal{B}^{\mathcal{T}}$. Furthermore, $\mathcal{M}$ can be reconstructed from this subset: for each transition $q \xrightarrow{i/o} r$ with $q \in \mathcal{B}^{\mathcal{T}}$ and $r \notin \mathcal{B}^{\mathcal{T}}$, seek a state p in $\mathcal{B}^{\mathcal{T}}$ that has the same behavior as r (*i. e.*, the transitions below p and r output the same symbols at the same time) and redirect the transition into $q \xrightarrow{i/o} p$. This is the idea behind $L^{\#}$: identify each state of $\mathcal{M}$ by constructing a finite tree via output queries. We call the data structure an *observation tree*.

3: In the sense that there exists a run from the root to a state of this subset that never leaves the subset.

Definition 3.3.3 (Observation tree). An *observation tree* $\mathcal{T}$ is a tree-shaped Mealy machine, *i. e.*, for every state $q \in Q^{\mathcal{T}}$ there exists a unique run $q_0^{\mathcal{T}} \xrightarrow{w} q$.

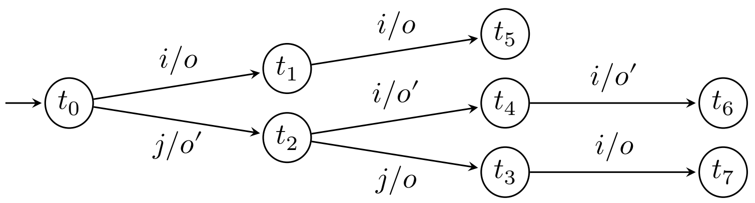


Figure 3.9: A sample observation tree $\mathcal{T}$.

Let us start by explaining how one can add new nodes and transitions in $\mathcal{T}$. Assume we already have $q_0^{\mathcal{T}} \xrightarrow{w} q \in \mathit{runs}(\mathcal{T})$ and we want to have a run reading $w \cdot v$. We thus need to add new nodes after q such that the transitions read the word v. For MMs, this is easy to achieve: simply ask $\mathbf{OQ}(w \cdot v)$ and add nodes and transitions accordingly (*i.e.*, use the outputs obtained from the query). From now on, we assume that any output query automatically creates the nodes and their transitions.

We first characterize the fact that $\mathcal{T}$ stores some of the runs of the teacher's MM $\mathcal{M}$. Then, we introduce the notion of *apartness* that is used to prove that two states of $\mathcal{T}$ correspond to two different states in $\mathcal{M}$. Finally, we give the constraints the tree must satisfy for a hypothesis to be constructed, and the construction itself.

Functional simulation

Given the fact that $\mathcal{T}$ is constructed via output queries, it must be that every run of $\mathcal{T}$ mimics a run of $\mathcal{M}$. In particular, $\mathcal{T}$ and $\mathcal{M}$ output the same word for every input word that can be read within $\mathcal{T}$. That is, for all word w labeling a run of $\mathcal{T}$, we have $\mathit{output}^{\mathcal{M}}(w) = \mathit{output}^{\mathcal{T}}(w)$. However, due to the tree-shaped nature of $\mathcal{T}$ and its finiteness, not all runs of $\mathcal{M}$ are present in $\mathcal{T}$. This notion of "mimicry" is formally characterized via a *functional simulation*, which maps every state of $\mathcal{T}$ to some state of $\mathcal{M}$.[4] In particular, the initial state of $\mathcal{T}$ must be mapped with the initial state of $\mathcal{M}$. Then, the remaining states can be obtained by performing a depth- or breadth-first search, for instance (as long as the input and output pairs match).

4: In [Vaa+22], functional simulations are defined for any MM. We define it here for an observation tree to have the same approach as in Part IV.

Definition 3.3.4 (Functional simulation)**.** Let $\mathcal{M}$ be a complete MM and $\mathcal{T}$ be an observation tree. A *functional simulation* $f : \mathcal{T} \to \mathcal{M}$ is a map $f : Q^{\mathcal{T}} \to Q^{\mathcal{M}}$ with

- ▶ $f(q_0^{\mathcal{T}}) = q_0^{\mathcal{M}}$, and
- ▶ for all $q, q' \in Q^{\mathcal{T}}$, $i \in I$, and $o \in O$, $q \xrightarrow{i/o} q'$ implies $f(q) \xrightarrow{i/o} f(q')$.

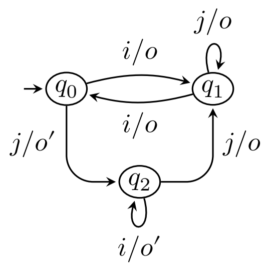
We say that $\mathcal{T}$ is an *observation tree for* $\mathcal{M}$ if there exists a functional simulation $f : \mathcal{T} \to \mathcal{M}$.

Example 3.3.5. Figure 3.9 gives an example of an observation tree $\mathcal{T}$. Again, it is a partial MM. So, transitions show the input and the output, separated by a slash.
Let $\mathcal{M}$ be the MM of Figure 2.3, which is repeated in the margin. Let us

show that $\mathcal{T}$ is an observation tree for $\mathcal{M}$, *i. e.*, that there exists a functional simulation $f : \mathcal{T} \to \mathcal{M}$. Let f such that

$$\begin{aligned} f(t_0) = f(t_5) = f(t_7) &= q_0 \\ f(t_1) = f(t_3) &= q_1 \\ f(t_2) = f(t_4) = f(t_6) &= q_2. \end{aligned}$$

One can then check that f is indeed a functional simulation.



Apartness of states

Thanks to the functional simulation $f : \mathcal{T} \to M$, we know that each state of $\mathcal{T}$ corresponds to a state of $\mathcal{M}$. Hence, if we observe a difference in behavior between two states p_0 and p_0' of $\mathcal{T}$, then this difference also exists between $f(p_0)$ and $f(p_0')$ in $\mathcal{M}$. In other words, if we have runs starting from p_0 and p_0' reading the same input word but outputting different output words, then $f(p_0)$ and $f(p_0')$ exhibit the same difference. Hence, we can conclude that p_0 and p_0' are *apart*, *i. e.*, they ought to be distinct.

Definition 3.3.6 (Apartness of states)**.** Two states p_0 and p_0' are *apart* if there are runs

$$p_0 \xrightarrow{i_1/o_1} p_1 \xrightarrow{i_2/o_2} \cdots \xrightarrow{i_n/o_n} p_n \in \mathit{runs}(\mathcal{T})$$

and

$$p_0' \xrightarrow{i_1/o_1'} p_1' \xrightarrow{i_2/o_2'} \cdots \xrightarrow{i_n/o_n'} p_n' \in \mathit{runs}(\mathcal{T})$$

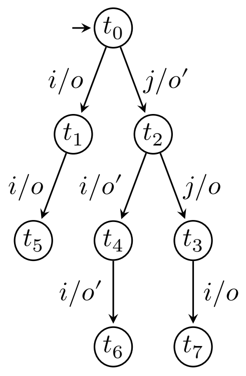
such that $o_n \neq o_n'$.
The word $w = i_1 \cdots i_n \in I^*$ is called a *witness* of $p_0 \# p_0'$, denoted by $w \vdash p_0 \# p_0'$.

Observe that $\#$ is symmetric, *i. e.*, if $q \# p$, then $p \# q$.

Example 3.3.7. Let $\mathcal{T}$ be the observation tree of Figure 3.9 (which is repeated in the margin) and $f : \mathcal{T} \to \mathcal{M}$ be the functional simulation of Example 3.3.5. Since $t_0 \xrightarrow{j/o'}$ and $t_2 \xrightarrow{j/o}$, we conclude that $j \vdash t_0 \# t_2$. Observe that $f(t_0) = q_0 \neq f(t_2) = q_2$, *i. e.*, it is correct to say that t_0 and t_2 are apart.
There may be multiple witnesses for the same apartness pair. For instance,

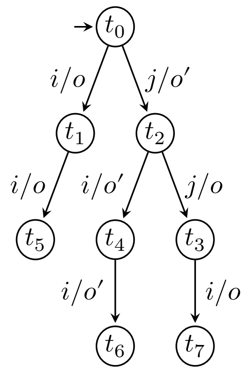
$$t_0 \xrightarrow{i/o} t_1 \xrightarrow{i/o} t_5 \qquad \text{and} \qquad t_2 \xrightarrow{i/o'} t_4 \xrightarrow{i/o'} t_6$$

imply that i and $i \cdot i$ are also witnesses of $t_0 \# t_2$.
Contrarily, we cannot conclude that $t_0 \# t_1$. Indeed, the only (nonempty) run from t_1 is $t_1 \xrightarrow{i/o} t_5$. As $t_0 \xrightarrow{i/o}$ outputs the same symbol, we are not able to distinguish t_1 and t_0. Hence, $\neg(t_0 \# t_1)$.
Finally, we give all possible apartness pairs with a possible witness for each



of them:

$$
\begin{array}{ll}
i \vdash t_0 \# t_2 & i \vdash t_0 \# t_4 \\
i \vdash t_1 \# t_2 & i \vdash t_1 \# t_4 \\
i \vdash t_2 \# t_3 & i \vdash t_3 \# t_4.
\end{array}
$$



Let us now claim two important lemmas, called the *weak co-transitivity* and *soundness* lemmas. The first one is heavily used in the learning algorithm to find new words to add in the tree to get more apartness pairs, while the second one states that the definition of apartness makes sense with regards to the functional simulation.

We start with the *weak co-transitivity lemma*: if we can read the witness of the apartness $p_0 \# p'_0$ from a third state r_0, then we can conclude that $r_0 \# p_0$ or $r_0 \# p'_0$ (or both). Hence, during the learning algorithm, if we need to deduce that $r_0 \# p_0$ or $r'_0 \# p'_0$ with $w \vdash p_0 \# p'_0$, it is sufficient to add transitions in $\mathcal{T}$ in order to obtain $r_0 \xrightarrow{w} \in runs(\mathcal{T})$. By this lemma, we will necessarily get (at least) one of the apartness states.

Lemma 3.3.8 (Co-transitivity lemma [Vaa+22]). *For any states p_0, p'_0, r_0 of $\mathcal{T}$ and word $w \in I^*$ such that*

- $w \vdash p_0 \# p'_0$, *and*
- $r_0 \xrightarrow{w} \in runs(\mathcal{M})$,

it holds that $w \vdash r_0 \# p_0$ or $w \vdash r_0 \# p'_0$.

Finally, the *soundness lemma*: as argued above, any $p \# p'$ occurring in $\mathcal{T}$ also proves that $f(p) \neq f(p')$. That is, the definition of apartness makes sense and is sound.

Lemma 3.3.9 (Soundness [Vaa+22]). *For a functional simulation $f : \mathcal{T} \to \mathcal{M}$,*

$$\forall p, p' \in Q^{\mathcal{T}} : p \# p' \Rightarrow f(p) \neq f(p').$$

Basis, frontier, and hypothesis construction

The objective of the learning process is to extend the tree in order to eventually construct a hypothesis $\mathcal{H}$. More precisely, we want to derive $\mathcal{H}$ immediately from $\mathcal{T}$, in the sense that the set of states of $\mathcal{H}$ is a subset of the states of $\mathcal{T}$. In order for this to make sense, we want to select a tree-shaped subset, which will be denoted $\mathcal{B}^{\mathcal{T}}$, of $\mathcal{T}$ that is rooted at $q_0^{\mathcal{T}}$. We then need to redirect the transitions leaving $\mathcal{B}^{\mathcal{T}}$ back to some state in $\mathcal{B}^{\mathcal{T}}$. We call this *folding* the tree.

We first properly characterize this aforementioned subset $\mathcal{B}^{\mathcal{T}}$ and give the constraints $\mathcal{T}$ has to satisfy for $\mathcal{H}$ to be constructed. In short, $\mathcal{B}^{\mathcal{T}}$ is composed of all the states p such that p is apart from any other state p' in $\mathcal{B}^{\mathcal{T}}$. By Lemma 3.3.9, we then know that p and p' necessarily represent two distinct states in $\mathcal{M}$. Since this holds for any p and p' in $\mathcal{B}^{\mathcal{T}}$, it follows that $\mathcal{B}^{\mathcal{T}}$ is comprised of the states that have been fully identified, *i. e.*, we know that they all correspond to distinct states in $\mathcal{M}$.

Definition 3.3.10 (Basis and frontier)**.** The states of $\mathcal{T}$ are split into three disjoint subsets:

- The *basis* $\mathcal{B}^{\mathcal{T}}$ is a subtree of $Q^{\mathcal{T}}$ such that $q_0^{\mathcal{T}} \in \mathcal{B}^{\mathcal{T}}$, and, for all $p \neq p' \in \mathcal{B}^{\mathcal{T}}, p \# p'$.
- The *frontier* $\mathcal{F}^{\mathcal{T}}$ is the set of immediate non-basis successors of basis states, *i. e.*,

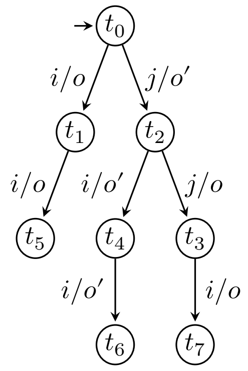
  $$\mathcal{F}^{\mathcal{T}} = \{r \in Q^{\mathcal{T}} \setminus \mathcal{B}^{\mathcal{T}} \mid \exists p \in \mathcal{B}^{\mathcal{T}} : p \rightarrow r\}.$$

  We say that $p \in \mathcal{B}^{\mathcal{T}}$ and $r \in \mathcal{F}^{\mathcal{T}}$ are *compatible* if $\neg(p \# r)$. We write $\mathit{compat}^{\mathcal{T}}(r)$ for the set of all such states p:

  $$\mathit{compat}^{\mathcal{T}}(r) = \{p \in \mathcal{B}^{\mathcal{T}} \mid \neg(p \# r)\}.$$
- The remaining states $Q^{\mathcal{T}} \setminus \left(\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}\right)$.

Example 3.3.11. Let us continue Example 3.3.7, *i. e.*, we define the basis and the frontier of the observation tree $\mathcal{T}$ of Figure 3.9 (which is repeated in the margin). The initial state t_0 is necessarily a basis state. Moreover, by the apartness pairs shown in Example 3.3.7, we can fix $\mathcal{B}^{\mathcal{T}} = \{t_0, t_2\}$. So, we have $\mathcal{F}^{\mathcal{T}} = \{t_1, t_3, t_4\}$ and the following compatible sets:

$$\mathit{compat}^{\mathcal{T}}(t_1) = \mathit{compat}^{\mathcal{T}}(t_3) = \{t_0\}, \qquad \mathit{compat}^{\mathcal{T}}(t_4) = \{t_2\}.$$



Remark 3.3.12. We highlight that, for a given observation tree $\mathcal{T}$, there may be multiple different bases. That is, there may be different ways to partition the states of $\mathcal{T}$.

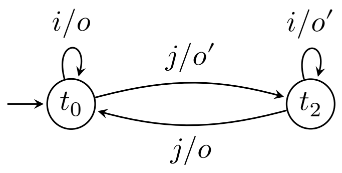
During the learning algorithm, we will ensure that the basis and the frontier satisfy the following constraints:

- the basis is *complete*, in the sense that $p \xrightarrow{i}$ is defined for every $i \in I$, and
- for every $r \in \mathcal{F}^{\mathcal{T}}$, $\mathit{compat}^{\mathcal{T}}(r) \neq \emptyset$.

That is, the outgoing transitions of every basis state are all defined, and each frontier state has a compatible state. This allows us to define the hypothesis in a straightforward way: for each frontier state r, pick one basis state q in $\mathit{compat}^{\mathcal{T}}(r)$ and redirect the transition $p \xrightarrow{i} r \in \mathit{runs}(\mathcal{T})$ into a transition $p \xrightarrow{i} q \in \mathit{runs}(\mathcal{H})$.

Definition 3.3.13 (Hypothesis construction)**.** Let $\mathfrak{h} : \mathcal{F}^{\mathcal{T}} \rightarrow \mathcal{B}^{\mathcal{T}}$ be a function such that, for all $r \in \mathcal{F}^{\mathcal{T}}$, $\mathfrak{h}(r) \in \mathit{compat}^{\mathcal{T}}(r)$. Then, $\mathcal{H}$ is an MM defined as follows:

- $Q^{\mathcal{H}} = \mathcal{B}^{\mathcal{T}}$,
- $q_0^{\mathcal{H}} = q_0^{\mathcal{T}}$, and
- all transitions that remain within the basis are copied as-is, while transitions entering a frontier state are redirected using $\mathfrak{h}$, *i. e.*, δ is



$$\begin{tikzcd} \arrow[r] & t_0 \arrow[loop above, "i/o"] \arrow[r, bend left, "j/o'"] & t_2 \arrow[loop above, "i/o'"] \arrow[l, bend left, "j/o"] \end{tikzcd}$$

Figure 3.10: The hypothesis constructed from the observation tree of Figure 3.9.

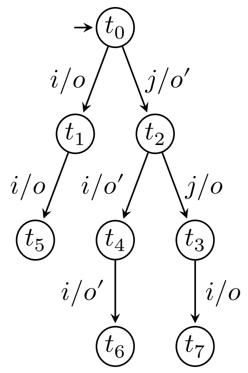
defined such that for all $p \xrightarrow{i/o} q \in \mathit{runs}(\mathcal{T})$ with $p \in \mathcal{B}^{\mathcal{T}}$ we have

- $p \xrightarrow{i/o} q \in \mathit{runs}(\mathcal{H})$ when $q \in \mathcal{B}^{\mathcal{T}}$, and
- $p \xrightarrow{i/o} \mathfrak{h}(q) \in \mathit{runs}(\mathcal{H})$ when $q \in \mathcal{F}^{\mathcal{T}}$

Example 3.3.14. Let us continue Example 3.3.11, *i.e.*, we consider the observation tree $\mathcal{T}$ of Figure 3.9 (again, the tree is repeated in the margin), with $\mathcal{B}^{\mathcal{T}} = \{t_0, t_2\}$ and $\mathcal{F}^{\mathcal{T}} = \{t_1, t_4, t_5\}$. Since the basis is complete and each compatible set is not empty, we can construct a hypothesis. As the compatible set of each frontier state contains a single basis state, defining $\mathfrak{h}$ is easy:

$$\mathfrak{h}(t_1) = \mathfrak{h}(t_3) = t_0, \qquad \mathfrak{h}(t_4) = t_2.$$

We thus construct the hypothesis MM given in Figure 3.10.



$$\begin{tikzcd} & & t_0 \arrow[dl, "i/o"'] \arrow[dr, "j/o'"] & & \\ & t_1 \arrow[dl, "i/o"'] & & t_2 \arrow[dl, "i/o'"'] \arrow[dr, "j/o"] & \\ t_5 & & t_4 \arrow[d, "i/o'"'] & & t_3 \arrow[d, "i/o"] \\ & & t_6 & & t_7 \end{tikzcd}$$

Observe that a unique hypothesis can be constructed if and only if each compatible set contains a unique element. Hence, in order to reduce the number of equivalence queries needed, $L^{\#}$ will also reduce the size of the compatible sets as much as possible.

3.3.2. Main loop

We now give the main loop of $L^{\#}$ for MMs. A pseudo-code is given in Algorithm 3.2. We initialize $\mathcal{T}$ to only contain $q_0^{\mathcal{T}}$ with $\mathcal{B}^{\mathcal{T}} = \{q_0^{\mathcal{T}}\}$, and $\mathcal{F}^{\mathcal{T}} = \emptyset$. The main loop is split into two parts:

Refinement loop The *refinement loop* extends the tree until the basis is complete, and $|\mathit{compat}^{\mathcal{T}}(r)| = 1$ for every frontier state r. To do so, it performs the following operations, in this order, until no more changes are possible:

Promotion If $\mathit{compat}^{\mathcal{T}}(r)$ is empty for some frontier state r, then we know that $q \# r$ for every $q \in \mathcal{B}^{\mathcal{T}}$. So, r can be added to $\mathcal{B}^{\mathcal{T}}$. We say that we *promote* r.

Completion If an i-transition is missing from some basis state p, we complete the basis with that transition by asking an output query.

WCT where **WCT** stands for Weak Co-Transitivity. In order to reduce the compatible sets as much as possible, we leverage Lemma 3.3.8 as follows. If we have some $r \in \mathcal{F}^{\mathcal{T}}$ and $p, p' \in \mathit{compat}^{\mathcal{T}}(r)$ such that $p \neq p'$ (*i.e.*, $\neg(p \# r)$ and $\neg(p' \# r)$), we know that there exists a witness w of $p \# p'$. By Lemma 3.3.8, if $r \xrightarrow{w} \in \mathit{runs}(\mathcal{T})$, it follows

Lemma 3.3.8. For any states p_0, p_0', r_0 of $\mathcal{T}$ and word $w \in I^*$ such that

- $w \vdash p_0 \# p_0'$, and
- $r_0 \xrightarrow{w} \in \mathit{runs}(\mathcal{M})$,

it holds that $w \vdash r_0 \# p_0$ or $w \vdash r_0 \# p_0'$.

Algorithm 3.2: The $L^{\#}$ algorithm for MMs.

```
1:  Initialize $\mathcal{T}$ with $\mathcal{B}^{\mathcal{T}} = \{q_0^{\mathcal{T}}\}$ and $\mathcal{F}^{\mathcal{T}} = \emptyset$
2:  while true do
3:      while $\mathcal{T}$ is changed do                                              ▷ Refinement loop
4:          if $\exists r \in \mathcal{F}^{\mathcal{T}}$ such that $compat^{\mathcal{T}}(r) = \emptyset$ then                    ▷ Promotion
5:              $\mathcal{B}^{\mathcal{T}} \leftarrow \mathcal{B}^{\mathcal{T}} \cup \{r\}$
6:              $\mathcal{F}^{\mathcal{T}} \leftarrow \mathcal{F}^{\mathcal{T}} \setminus \{r\}$
7:          else if $\exists p \in \mathcal{B}^{\mathcal{T}}, i \in I$ such that $p \xrightarrow{i} \notin runs(\mathcal{T})$ then   ▷ Completion
8:              Let $v$ be the word such that $q_0^{\mathcal{T}} \xrightarrow{v} p \in runs(\mathcal{T})$
9:              OQ$(v \cdot i)$
10:             $\mathcal{F}^{\mathcal{T}} \leftarrow \mathcal{F}^{\mathcal{T}} \cup \{r \mid p \xrightarrow{i} r\}$
11:         else if $\exists r \in \mathcal{F}^{\mathcal{T}}, p \neq p' \in compat^{\mathcal{T}}(r)$ then                      ▷ WCT
12:             Let $v$ be the word such that $q_0^{\mathcal{T}} \xrightarrow{v} r \in runs(\mathcal{T})$
13:             Let $w$ be a witness of $p \# p'$
14:             OQ$(v \cdot w)$
15:     $\mathcal{H} \leftarrow$ CONSTRUCTHYPOTHESIS                    ▷ Once $\mathcal{B}^{\mathcal{T}}$ and $\mathcal{F}^{\mathcal{T}}$ are stabilized
16:     $v \leftarrow$ EQ$(\mathcal{H})$
17:     if $v =$ yes then return $\mathcal{H}$ else PROCCOUNTEREX$(v)$
```

that $p \# r \vee p' \# r$. So, we extend the tree by adding w from r and necessarily obtain that $compat^{\mathcal{T}}(r)$ is smaller.

Hypothesis and equivalence Once the refinement step no longer modifies $\mathcal{T}$, we can construct a hypothesis $\mathcal{H}$ from $\mathcal{T}$ and call **EQ**$(\mathcal{H})$. If the teacher answers **yes**, we then return $\mathcal{H}$. Otherwise, a counterexample w is provided and can be used to extend $\mathcal{T}$, before refining it again.

Counterexample processing

Let $w = i_1 \cdots i_n$ be the counterexample returned by some **EQ**$(\mathcal{H})$. We explain how to extend the tree such that $\mathcal{H}$ can no longer be constructed. Since w is a counterexample, there must be a transition in $\mathcal{H}$ that is incorrect, in the sense that it should lead to a state that is not yet known, *i.e.*, we defined $p \xrightarrow{i} \mathfrak{h}(r) \in runs(\mathcal{H})$ but r should be a basis state. In order to see this, we will add new nodes in $\mathcal{T}$ until p is no longer in $compat^{\mathcal{T}}(r)$. As we already ensured that $compat^{\mathcal{T}}(r) = \{p\}$, it follows that r will be promoted at the start of the next refinement loop.

Here, we give an algorithm that is different from the one proposed in [Vaa+22] but is the foundation of the counterexample processing for Mealy machines with timers of Part IV. Importantly, the one we describe is *less* efficient than Vaandrager *et al.*'s approach. While [Vaa+22] performs a binary search to cut the counterexample in half, we do a more simple linear search. We thus, in general, require more output queries and the number of symbols used in them is higher.[5]

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

5: In particular, this means that we do not obtain the complexity claimed in Theorem 3.3.2.

First of all, observe that if we add $q_0^{\mathcal{T}} \xrightarrow{w} q$ to the tree, it must be that $q \notin \mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}$. By construction, it is impossible to have a mistake with the transitions that remain within the basis. So, the mistake must come from a (potentially missing) transition $p \xrightarrow{i}$ with $p \notin \mathcal{B}^{\mathcal{T}}$. The goal of the counterexample processing is thus to identify one such problematic transition.

Notice that w may be very long. By analyzing *output*$^{\mathcal{M}}(w)$ (obtained by an output query) and *output*$^{\mathcal{H}}(w)$, it is possible to find a prefix $w' \cdot i$ of w such that $\mathcal{M}$ and $\mathcal{H}$ do not output the same symbol when reading i. We then add $w' \cdot i$ in the tree and obtain

$$q_0^{\mathcal{T}} \xrightarrow{w'} q \xrightarrow{i/o} \in \mathit{runs}(\mathcal{T}) \wedge q_0^{\mathcal{H}} \xrightarrow{w'} q' \xrightarrow{i/o'} \in \mathit{runs}(\mathcal{H}) \wedge o \neq o'.$$

However, adding $q_0^{\mathcal{T}} \xrightarrow{w' \cdot i}$ in the tree may not be enough for a compatible set to shrink. Let $r_1 \in \mathcal{F}^{\mathcal{T}}$ and $w' \cdot i = v_1 \cdot v_1'$ such that $v_1' \neq \varepsilon$ and

$$q_0^{\mathcal{T}} \xrightarrow{v_1} r_1 \xrightarrow{v_1'} \in \mathit{runs}(\mathcal{T}).$$

It may be that we cannot read v_1' from $\mathfrak{h}(r_1)$ (recall that $\mathfrak{h}(r_1)$ belongs to *compat*$^{\mathcal{T}}(r_1)$), *i. e.*, $\mathfrak{h}(r_1) \xrightarrow{v_1'} \notin \mathit{runs}(\mathcal{T})$. In that case, we ask an output query to be able to read v_1' from $\mathfrak{h}(r_1)$.

If we obtain $r_1 \# \mathfrak{h}(r_1)$, we can stop. Otherwise, we can apply the same principle: let $r_2 \in \mathcal{F}^{\mathcal{T}}$ and $v_1' = v_2 \cdot v_2'$ such that

$$q_0^{\mathcal{T}} \xrightarrow{v_2} r_2 \xrightarrow{v_2'} \in \mathit{runs}(\mathcal{T}),$$

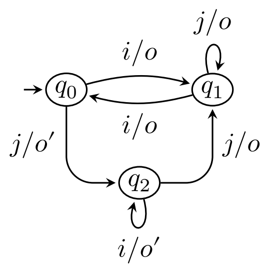
and we ask an output query to add $\mathfrak{h}(r_2) \xrightarrow{v_2'} \in \mathit{runs}(\mathcal{T})$. If we have $r_2 \# \mathfrak{h}(r_2)$, we can stop. Otherwise, we split v_2', extend the tree, and so on until we obtain $v_j \# \mathfrak{h}(v_j)$ for some j.

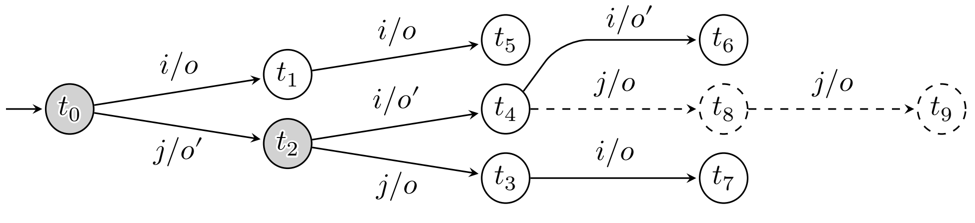
We give an example of this processing, before claiming that such a j always exists.

Example 3.3.15. Let $\mathcal{H}$ be the MM of Figure 3.10 and $\mathcal{M}$ be the MM of the teacher given in Figure 2.3 (which is repeated in the margin). Notably, in Example 3.3.14, $\mathcal{H}$ was constructed with the function $\mathfrak{h}$ such that

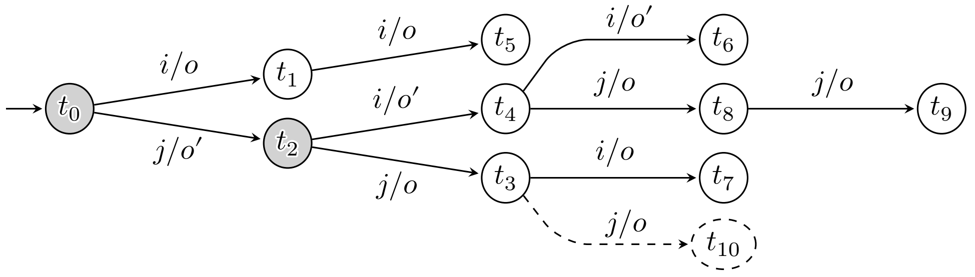
$$\mathfrak{h}(t_1) = \mathfrak{h}(t_3) = t_0, \qquad \mathfrak{h}(t_4) = t_2.$$

We can see that $\mathcal{M}$ and $\mathcal{H}$ are not equivalent due to the counterexample $w = j{\cdot}i{\cdot}j{\cdot}j$. Indeed, *output*$^{\mathcal{M}}(w) = o'{\cdot}o'{\cdot}o{\cdot}o$ while *output*$^{\mathcal{H}}(w) = o'{\cdot}o'{\cdot}o{\cdot}o'$. We thus add $t_0 \xrightarrow{j \cdot i \cdot j \cdot j}$ in $\mathcal{T}$ by asking **OQ**. Figure 3.11a gives the resulting observation tree.
Notice that $\neg(t_4 \# t_2)$, *i. e.*, we can still construct $\mathcal{H}$ and we need to extend the tree further. We split $w = (j \cdot i) \cdot (j \cdot j)$ as $t_0 \xrightarrow{j \cdot i} t_4$ and $t_4 \in \mathcal{F}^{\mathcal{T}}$. We add $j \cdot j$ from $\mathfrak{h}(t_4) = t_2$, *i. e.*, we now have the run $t_2 \xrightarrow{j \cdot j} t_{10}$, as shown in Figure 3.11b.
This time, we have $j \vdash t_3 \# t_0$, meaning that t_0 is no longer compatible with t_3. That is, *compat*$^{\mathcal{T}}(t_3)$ is now empty and t_3 can be promoted. Hence, $\mathcal{H}$ can no longer be constructed from the resulting tree.





(a) After adding the complete counterexample.



(b) After adding $j \cdot j$ from t_2.

Figure 3.11: Observation trees obtained by processing the counterexample $j \cdot i \cdot j \cdot j$. Newly added nodes and transitions are drawn with dashed lines. Basis nodes are highlighted with a gray background.

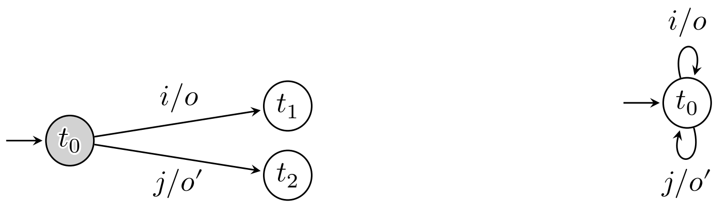
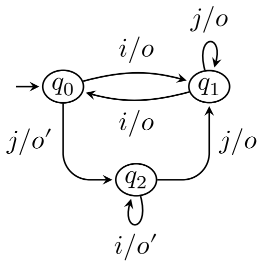


Figure 3.12: The observation tree $\mathcal{T}_1$ and the hypothesis $\mathcal{H}_1$ constructed from it. Basis states are highlighted with a gray background.

While this counterexample processing algorithm differs from the one presented in [Vaa+22], it is not hard to adapt their proof to show that our algorithm is sufficient to break the last hypothesis.

Lemma 3.3.16. *There exists a frontier state r such that $r \# \mathfrak{h}(r)$ after processing a counterexample w.*

3.3.3. Complete example



Finally, we give an example of a complete run of $L^{\#}$. Let $\mathcal{M}$ be the MM of Figure 2.3 (which is repeated in the margin). Similarly to the complete example of L^* (Section 3.2.3), we enumerate each observation tree and the basis and frontier states, *e.g.*, $\mathcal{T}_1, \mathcal{B}^{\mathcal{T}_3}, \mathcal{F}^{\mathcal{T}_4}$, and so on.

The initial observation tree has a single state t_0, which is in the basis. Since $I = \{i, j\}$, the basis is not complete and we can apply **Completion** twice (once for i and once for j). The resulting tree $\mathcal{T}_1$ is given in Figure 3.12. We have $\mathcal{B}^{\mathcal{T}_1} = \{t_0\}, \mathcal{F}^{\mathcal{T}_1} = \{t_1, t_2\}$, and $compat^{\mathcal{T}_1}(t_1) = compat^{\mathcal{T}_1}(t_2) = t_0$. That is, we satisfy every condition to construct a hypothesis $\mathcal{H}_1$, which is also given in Figure 3.12.

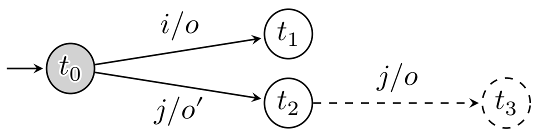


Figure 3.13: The observation tree $\mathcal{T}_2$. Newly added nodes and transitions are drawn with dashed lines.

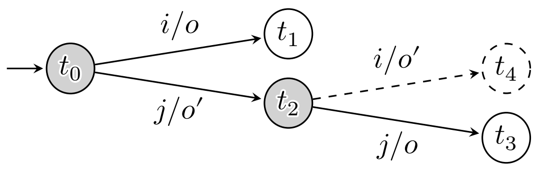


Figure 3.14: The observation tree $\mathcal{T}_3$.

Clearly, $\mathcal{H}_1$ and $\mathcal{M}$ are not equivalent. Let $w_1 = j \cdot j \cdot j$ be the counterexample returned by **EQ**$(\mathcal{H}_1)$, yielding

$$\mathit{output}^{\mathcal{M}}(j \cdot j \cdot j) = o' \cdot o \cdot o,$$

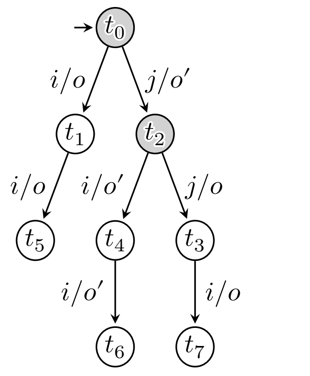
and

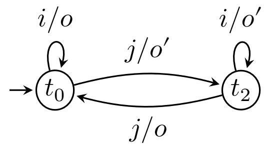
$$\mathit{output}^{\mathcal{H}_1}(j \cdot j \cdot j) = o' \cdot o' \cdot o'.$$

That is, the prefix $j \cdot j$ is sufficient to distinguish $\mathcal{H}_1$ and $\mathcal{M}$. We thus add $j \cdot j$ to the tree and obtain $\mathcal{T}_2$ as shown in Figure 3.13. Observe that $\mathit{compat}^{\mathcal{T}_2}(t_1) = \{t_0\}$ and $\mathit{compat}^{\mathcal{T}_2}(t_2) = \emptyset$, *i.e.*, we do not need to process the counterexample anymore. We can then promote t_2 (as its compatibility set is empty) and complete the new basis to obtain $\mathcal{T}_3$ from Figure 3.14. This time, we have

$$\mathit{compat}^{\mathcal{T}_3}(t_1) = \mathit{compat}^{\mathcal{T}_3}(t_3) = \mathit{compat}^{\mathcal{T}_3}(t_4) = \{t_0, t_2\}.$$

We thus apply **WCT** three times. First, we minimize the set $\mathit{compat}^{\mathcal{T}_3}(t_1)$ by adding the witness i of $t_0 \# t_2$ from the state t_1. We obtain that $t_1 \# t_2$, as $t_1 \xrightarrow{i/o}$ and $t_2 \xrightarrow{i/o'}$. We also add the witness i from t_3 and t_4 and obtain $t_3 \# t_2$ and $t_4 \# t_0$. The resulting tree is given in Figure 3.9 which is repeated in the margin (except that basis states are highlighted). As explained in Example 3.3.14, we construct the hypothesis $\mathcal{H}_2$ from Figure 3.10, which is also repeated in the margin, below the observation tree.





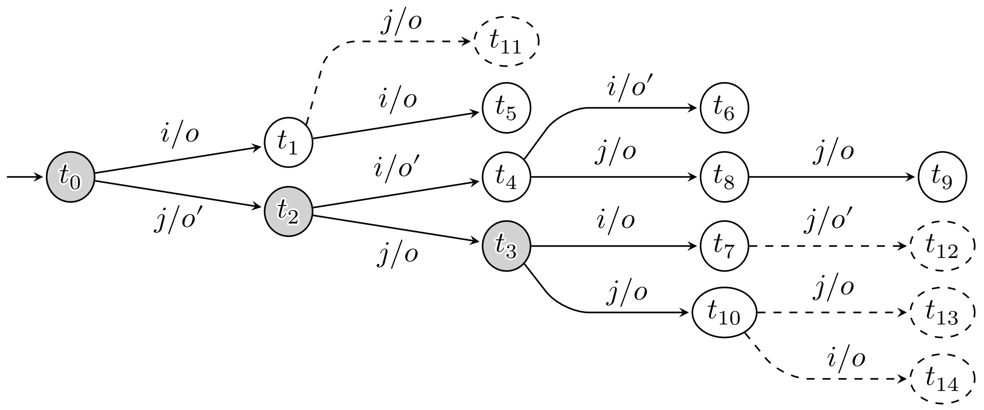
Let $w = j \cdot i \cdot j \cdot j$ be the counterexample returned by **EQ**$(\mathcal{H}_2)$. We process it as explained in Example 3.3.15 and obtain the tree $\mathcal{T}_4$ of Figure 3.11b. We have

$$\mathit{compat}^{\mathcal{T}_4}(t_1) = \{t_0\},$$
$$\mathit{compat}^{\mathcal{T}_4}(t_3) = \emptyset,$$

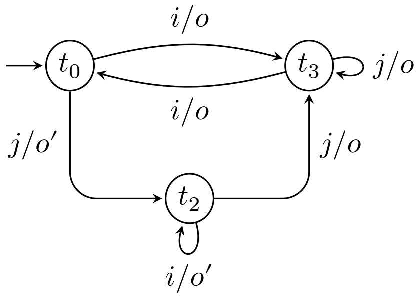
and

$$\mathit{compat}^{\mathcal{T}_4}(t_4) = \{t_2\}.$$

We can thus promote t_3. As $t_3 \xrightarrow{i}$ and $t_3 \xrightarrow{j}$ are both already defined, we do not have to apply **Completion**. That is, $\mathcal{T}_5$ is exactly $\mathcal{T}_4$, except that $t_3 \in \mathcal{B}^{\mathcal{T}_5}$,

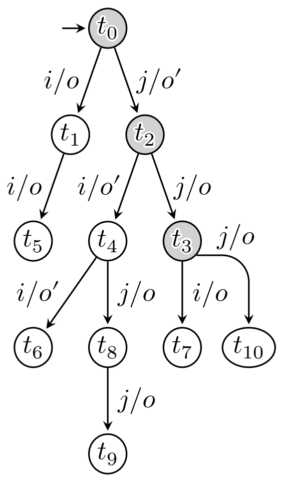


(a) The tree $\mathcal{T}_6$.



(b) The hypothesis $\mathcal{H}_6$.

Figure 3.15: The observation tree $\mathcal{T}_6$ and its hypothesis $\mathcal{H}_6$.



and is given in the margin. This results in the following compatible sets:

$$\begin{aligned} compat^{\mathcal{T}_5}(t_1) &= \{t_0, t_3\}, \\ compat^{\mathcal{T}_5}(t_4) &= \{t_2\}, \end{aligned}$$

and

$$compat^{\mathcal{T}_5}(t_7) = compat^{\mathcal{T}_5}(t_{10}) = \{t_0, t_2, t_3\}.$$

We thus apply **WCT** to reduce the sets. Since $j \vdash t_0 \# t_3$, we add a transition reading j from t_1, t_7, and t_{10}. We also need to add a transition reading i from t_{10}, as $i \vdash t_2 \# t_3$. We then obtain the observation tree $\mathcal{T}_6$ of Figure 3.15a with the following sets:

$$\begin{aligned} compat^{\mathcal{T}_6}(t_1) = compat^{\mathcal{T}_6}(t_{10}) &= \{t_3\}, \\ compat^{\mathcal{T}_6}(t_4) &= \{t_2\}, \end{aligned}$$

and

$$compat^{\mathcal{T}_6}(t_7) = \{t_0\}.$$

Since $\mathcal{T}_6$ can no longer be refined, we can construct a hypothesis $\mathcal{H}_6$, given in Figure 3.15b. This time, **EQ**$(\mathcal{H}_6)$ returns **yes**, *i.e.*, we are done.

Part II.

Learning Realtime One-Counter Automata

One-Counter Automata and Learning Visibly One-Counter Automata 4.

Our focus in the second part *Learning Realtime One-Counter Automata* is to provide an active learning algorithm for a subfamily of *one-counter automata*, which are, in general, NFAs extended with a single natural counter. Such an automaton can trigger different transitions according to the current value of the counter. Since the counter does not have an upper bound, a one-counter automaton induces an infinite transition system, unlike NFAs and DFAs. This makes active learning algorithms for this class harder.

This chapter motivates why learning one-counter automata is interesting, and summarizes a learning algorithm for a specific subfamily of one-counter automata, called *visibly one-counter automata* [NL10]. That is, this chapter serves as an introduction to Chapter 5 and its Appendix A, which contain our main contributions to this topic.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

Chapter contents

4.1. Introduction

We introduced in Section 3.2 an active learning algorithm called L^* that can automatically infer DFAs by querying a *teacher* with *membership* and *equivalence queries* [Ang87]. As said in Chapter 1, an important application of active learning is to learn black-box models from (legacy) software and hardware systems [GPY06; PVY02]. However, handling real-world applications usually involves tailor-made abstractions to circumvent elements of the system which result in an infinite state space [Aar+15]. We thus need learning algorithms that focus on more expressive models, able to represent such infinite state spaces.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

[GPY06]: Groce et al. (2006), "Adaptive Model Checking"

[PVY02]: Peled et al. (2002), "Black Box Checking"

[Aar+15]: Aarts et al. (2015), "Generating models of infinite-state communication protocols using regular inference with abstraction"

In this part, we consider *one-counter automata*, which are finite automata extended with a natural *counter* that can be increased, decreased, and tested for equality against a finite number of values. The counter allows such automata to capture the behavior of some infinite-state systems. Additionally, their expressiveness has been shown sufficient to verify programs with lists [Bou+11] and validate XML streams [CR04]. To the best of our knowledge, there is no learning algorithm for general one-counter automata. Section 4.2 gives

[Bou+11]: Bouajjani et al. (2011), "Programs with lists are counter automata"

[CR04]: Chitic et al. (2004), "On Validation of XML Streams Using Finite State Machines" a general definition of deterministic one-counter automata and argues why learning them is complex.

Hence, we focus on two subfamilies of one-counter automata:

- *visibly one-counter automata* in which counter operations solely depend on the alphabet symbol read by the transition (*e.g.*, each time we read an a, we must increment the counter). These restrictions allowed Neider and Löding to design an active learning algorithm [NL10], for which we give a summary in Section 4.3.
- *realtime one-counter automata* which lift this restriction on the counter operation, *i.e.*, each transition can arbitrarily increment or decrement the counter no matter the read symbol. This family and its learning algorithm (which is our main contribution in this part) will be discussed in Chapter 5.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

In both cases, we assume that we can observe the counter value, either due to the sequence of input symbols read so far, or by querying the system we want to learn. This means that the system with which we interact is not really a black box. Rather, we see it as a gray box. Several recent active-learning works make such assumptions to learn complex languages or ameliorate query-usage bounds. For instance, in [Ber+21], the authors assume they have information about the target language L in the form of a superset of it. Similarly, in [AR16], the authors assume L is obtained as the composition of two languages, one of which they know in advance. In [MO20], the teacher is assumed to have an executable automaton representation of the (infinite-word) target language. This helps them learn the automaton directly and to do so more efficiently than other active learning algorithms for the same task. Finally, in [Gar+20] it is assumed that constraints satisfied along the run of a system can be made visible. They leverage this (tainting technique) to give a scalable learning algorithm for register automata.

[Ber+21]: Berthon et al. (2021), "Active Learning of Sequential Transducers with Side Information About the Domain"

[AR16]: Abel et al. (2016), "Gray-Box Learning of Serial Compositions of Mealy Machines"

[MO20]: Michaliszyn et al. (2020), "Learning Deterministic Automata on Infinite Words"

[Gar+20]: Garhewal et al. (2020), "Grey-Box Learning of Register Automata"

4.2. Deterministic one-counter automata

In general, a one-counter automaton is an NFA that is augmented with a counter. We focus here on *deterministic* one-counter automaton and refer to [FMR68; Pet11] for a more general definition. The counter can be tested for equality against a finite number of values when triggering a transition. In the context of this thesis, we restrict the tests to either check whether the counter value is zero, or is strictly greater than zero. That is, we consider $\{=0, >0\}$ as the set of possible *guards* on the transitions. Furthermore, we impose that the counter value can be changed by at most one, *i.e.*, the set of allowed *counter operations* is $\{+1, -1, 0\}$. However, unlike DFAs, we allow a one-counter automaton to have transitions that do not read any symbol, *i.e.*, ε-transitions.[1] This means that an automaton can still increment or decrement the counter an arbitrary number of times between two input symbols. That being said, we forbid the definition of any a-transition with $a \in \Sigma$ when an ε-transition is defined for a given state and a given guard. That is, either we define an ε-transition, or we define transitions reading actual symbols.

[FMR68]: Fischer et al. (1968), "Counter Machines and Counter Languages"

[Pet11]: Petersen (2011), "Simulations by Time-Bounded Counter Machines"

1: Allowing ε-transitions in DFAs does not change the expressivity.

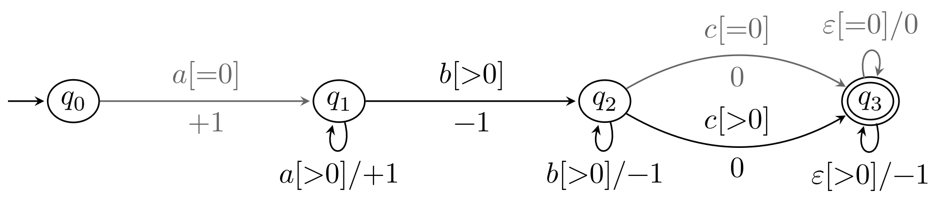


Figure 4.1: A sample deterministic one-counter automaton.

Deterministic one-counter automata can also be seen as deterministic pushdown automata in which the stack alphabet is composed of a unique symbol. The guard $=0$ (resp. >0) is then equivalent to checking whether the stack is empty (resp. not empty).

We first properly define what is a deterministic one-counter automaton, followed by its runs (where we do not yet consider the counter value, *i. e.*, we see the automaton as a DFA). In particular, we define *sound* automata that ensure that the counter can never be decremented when it is already zero, *i. e.*, in a sound automaton, the counter will never be negative. We then define the semantics of a DOCA, and argue, in Section 4.2.2, why it is hard to learn a DOCA (intuitively, this is due to the ε-transitions). Section 4.3 gives the main ideas behind an active learning algorithm for a subclass of DOCAs, which serves as a stepping stone for Chapter 5.

Definition 4.2.1 (Deterministic one-counter automata)**.** A *deterministic one-counter automaton* (*DOCA*, for short) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where:

- Σ is an alphabet,
- Q is a non-empty finite set of states, with $q_0 \in Q$ the initial state,
- $F \subseteq Q$ is the set of final states, and
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \{=0, >0\} \rightharpoonup Q \times \{+1, -1, 0\}$ is a transition function such that, for all $q \in Q, a \in \Sigma$, and $g \in \{=0, >0\}$, $\delta(q, a, g)$ can be defined only when $\delta(q, \varepsilon, g)$ is not.
  We write $q \xrightarrow[o]{a[g]} p$ when $(p, o) \in \delta(q, a, g)$.

A DOCA $\mathcal{A}$ is termed *sound* if for every $q \xrightarrow[o]{a[=0]}$ it holds that $o \neq -1$.

As for NFAs and DFAs, we add a superscript to indicate which automaton is considered, and missing symbols are quantified existentially, *e. g.*, $q \xrightarrow{a[g]} p$ means that there exist a state p and a counter operation o such that $q \xrightarrow[o]{a[g]} p$.

A *run* of $\mathcal{A}$ either consists of a single state p_0 or of a nonempty sequence of transitions

$$p_0 \xrightarrow{a_1[g_1]} p_1 \xrightarrow{a_2[g_2]} \cdots \xrightarrow{a_n[g_n]} p_n.$$

We denote by *runs*$(\mathcal{A})$ the set of runs of $\mathcal{A}$. As for finite automata, we often write $q \xrightarrow{a[g]} \in$ *runs*$(\mathcal{A})$ to highlight that $\delta(q, a, g)$ is defined. Note that any run is uniquely determined by its first state and its sequence of symbols and guards.

Example 4.2.2. A 4-state DOCA $\mathcal{A}$ over $\Sigma = \{a, b\}$ is given in Figure 4.1. Each transition gives the input symbol, the guard (between square brackets), and the counter operation (either after a slash or below the transition). In order to help the reading, transitions using the guard [=0] are drawn in gray.
A sample run is

$$q_0 \xrightarrow[+1]{a[=0]} q_1 \xrightarrow[+1]{a[>0]} q_1 \xrightarrow[-1]{b[>0]} q_2 \xrightarrow[0]{c[=0]} q_3 \xrightarrow[-1]{\varepsilon[>0]} q_3.$$

It is not hard to see that $\mathcal{A}$ is sound.

4.2.1. Semantics

Let us now define the semantics of a sound DOCA $\mathcal{A}$. Intuitively, we must keep track of the current state q and the current counter value n, in a pair we call a *configuration*. If n is zero, we then process the next symbol a by retrieving the pair $(p, o) = \delta(q, a, {=}0)$ (if it is defined), applying the counter operation o, and going to the state p. That is, we reach the configuration $(p, n + o)$. Likewise when $n > 0$, except that we pass >0 to δ. From the reached configuration $(p, n + o)$, we may process a new symbol, and so on. That is, the semantics of $\mathcal{A}$ are defined via a (potentially infinite) transition system. Recall that it is impossible to decrement a counter equal to zero, when $\mathcal{A}$ is sound.

Definition 4.2.3 (Counted runs)**.** Let $\mathcal{A}$ be a sound DOCA, $(q, n), (p, m) \in Q \times \mathbb{N}$ be two configurations, and $a \in \Sigma \cup \{\varepsilon\}$. There exists a transition $(q, n) \xrightarrow{a} (p, m)$ if and only if $m = n + o$ and

$$(p, o) = \begin{cases} \delta(q, a, {=}0) & \text{if } n = 0 \\ \delta(q, a, {>}0) & \text{if } n > 0. \end{cases}$$

A *counted run* of $\mathcal{A}$ is either a single configuration (p_0, n_0) or a nonempty sequence of transitions

$$(p_0, n_0) \xrightarrow{a_1} (p_1, n_1) \xrightarrow{a_2} \cdots \xrightarrow{a_\ell} (p_\ell, n_\ell).$$

We denote by *cruns*$(\mathcal{A})$ the set of all counted runs of $\mathcal{A}$.

Again, missing symbols in $(q, n) \xrightarrow{a} (p, m)$ are quantified existentially. We lift the notation to words as usual:

$$(p_0, n_0) \xrightarrow{a_1 \cdots a_\ell} (p_\ell, n_\ell) \in \textit{cruns}(\mathcal{A})$$

if there exists a counted run

$$(p_0, n_0) \xrightarrow{a_1} \cdots \xrightarrow{a_\ell} (p_\ell, n_\ell) \in \textit{cruns}(\mathcal{A}).$$

Notice that it is easy to go from a counted run to a run (by adding the appropriate guards and dropping the counter values). However, in general, given a run and a starting counter value, a corresponding counted run may not exist, as illustrated in Example 4.2.5.

Let us now define the language of $\mathcal{A}$ as the set of all words w that label a counted run from the *initial configuration* $(q_0, 0)$ to some configuration $(q, 0)$ with $q \in F$.

Definition 4.2.4 (Deterministic one-counter language)**.** The language accepted by a sound DOCA $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q \in F : (q_0, 0) \xrightarrow{w} (q, 0) \in \mathit{cruns}(\mathcal{A})\}.$$

We say that a language L is a *deterministic one-counter language* (*DOCL*, for short) if there is a sound DOCA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

Example 4.2.5. Let $\mathcal{A}$ be the DOCA of Figure 4.1, which is repeated in the margin. A sample counted run is

$$(q_0, 0) \xrightarrow{a} (q_1, 1) \xrightarrow{a} (q_1, 2) \xrightarrow{b} (q_2, 1) \xrightarrow{c} (q_3, 1) \xrightarrow{\varepsilon} (q_3, 0).$$

Its corresponding run is

$$q_0 \xrightarrow[+1]{a[=0]} q_1 \xrightarrow[+1]{a[>0]} q_1 \xrightarrow[-1]{b[>0]} q_2 \xrightarrow[0]{c[>0]} q_3 \xrightarrow[-1]{\varepsilon[>0]} q_3.$$

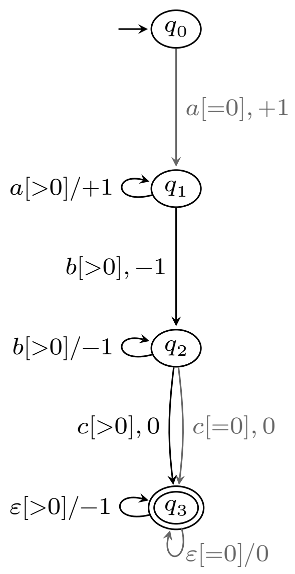
Observe that there is another counted run of $\mathcal{A}$ labeled by $a \cdot a \cdot b \cdot c$, obtained by not triggering the ε-transition:

$$(q_0, 0) \xrightarrow{a} (q_1, 1) \xrightarrow{a} (q_1, 2) \xrightarrow{b} (q_2, 1) \xrightarrow{c} (q_3, 1).$$

That is, a DOCA may still have multiple runs for a given word.
Since there exists a run of $\mathcal{A}$ reading $a \cdot a \cdot b \cdot c$ that ends in the configuration $(q_3, 0)$ (with $q_3 \in F$), we conclude that $a \cdot a \cdot b \cdot c$ is accepted by $\mathcal{A}$. One can show that the language of $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{a^n b^m c \mid 0 < m \leq n\}.$$

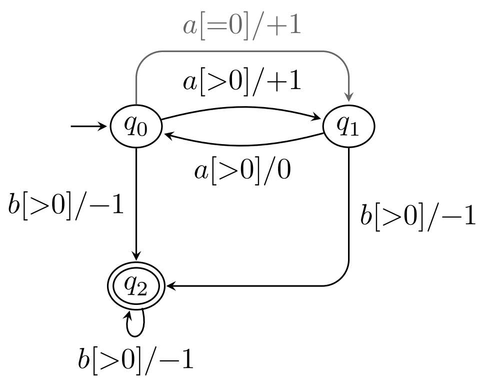
Finally, the run of Example 4.2.2 has no corresponding counted run. Indeed, the guard $=0$ cannot be satisfied when reading c (as the counter value is necessarily strictly greater than zero).



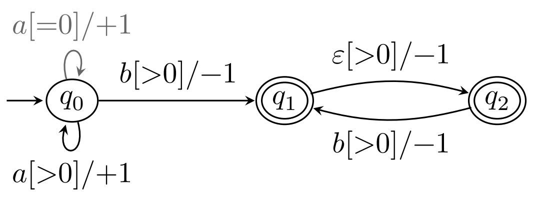
4.2.2. Problems for active learning

To the best of our knowledge, there is no active learning algorithm for DOCLs. In order to give the intuition as to why learning DOCLs is a hard task, let us consider the language $L = \{a^n b^{\lceil n/2 \rceil} \mid 0 < n\}$. Figure 4.2 gives two DOCAs accepting L. We highlight two problems, with regards to learning:

- The presence of ε-transitions in the DOCA of Figure 4.2b means that the DOCA may have multiple (counted) runs for a given word. For instance, reading the word $a \cdot a \cdot b$ may end in the configuration $(q_1, 1)$ or $(q_2, 0)$. That is, even if the automaton is deterministic (in the sense that, when



(a) The DOCA $\mathcal{A}_1$.



(b) The DOCA $\mathcal{A}_2$.

Figure 4.2: Two DOCAs accepting the language $L = \{a^n b^{\lceil n/2 \rceil} \mid 0 < n\}$.

a transition is triggered, there is at most one possible transition), the behavior of a DOCA is still somewhat non-deterministic as, in the worst case, some runs can have arbitrary lengths. In other words, a DOCA can potentially have an infinite number of runs for a given word. This non-determinism may be avoided by changing the syntax and semantics of the model.[2]

▶ More importantly, two different DOCAs may compute two different counter values for a given word. For instance, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the DOCAs of Figure 4.2. While they both accept the same language, they have two different approaches: $\mathcal{A}_1$ increases the counter value every other a (*i.e.*, the counter stores the number of b symbols to read), while $\mathcal{A}_2$ increases it every a and relies on ε-transitions to decrement the counter adequately while processing the b symbols. Hence, we have

$$(q_0^{\mathcal{A}_1}, 0) \xrightarrow{a} (q_1^{\mathcal{A}_1}, 1) \xrightarrow{a} (q_0^{\mathcal{A}_1}, 1) \xrightarrow{a} (q_1^{\mathcal{A}_1}, 2) \in \mathit{cruns}(\mathcal{A}_1)$$

and

$$(q_0^{\mathcal{A}_2}, 0) \xrightarrow{a} (q_0^{\mathcal{A}_2}, 1) \xrightarrow{a} (q_0^{\mathcal{A}_2}, 2) \xrightarrow{a} (q_0^{\mathcal{A}_2}, 3) \in \mathit{cruns}(\mathcal{A}_2).$$

That is, both automata do not associate the same counter value to the word $a \cdot a \cdot a$. This implies two points:

- The counter value of a given word depends on the DOCA, *i.e.*, it is a property of the automaton, not of the language (and there is no trivial canonic counter value one could compute). That is, the teacher has to know a DOCA and must propose a new kind of query to retrieve a counter value for a word.[3]
- It is not easy to deduce the counter operations occurring within a DOCA.

These reasons make inferring a correct behavior for the counter a hard task. Hence, Fahmy and Roos studied learning DOCAs where ε-transitions are not allowed [FR95]. We will discuss their paper in more details in the next chapter. Neider and Löding also designed a learning algorithm for a further restriction of DOCAs where the counter operations are deduced from the read symbols [NL10], which is the focus of the next section.

2: For instance, one may forbid any ε-transition leaving an accepting configuration, and request that we only consider "maximal" counted runs (in the sense that every possible ε-transition is triggered at the end of a counted run).

3: This will be properly introduced in Chapter 5.

[FR95]: Fahmy et al. (1995), "Efficient Learning of Real Time One-Counter Automata"

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

4.3. Learning visibly one-counter automata

In short, the restrictions Neider and Löding consider in [NL10] are:

- each transition has to read an input symbol, and
- every time we read a symbol a, we must perform the same counter operation. That is, the modification of the counter solely depends on the input symbol.

These restrictions are akin to those of *visibly pushdown automata* from [AM04]. We thus call these *visibly one-counter automata*. In this section, we give the main ideas behind Neider and Löding's active learning algorithm for visibly one-counter automata [NL10], which gives intuition and serves as a subroutine for our learning algorithm presented in the next chapter. In particular, we only provide proofs here when they are useful to build intuition.

[AM04]: Alur et al. (2004), "Visibly pushdown languages"

We first define visibly one-counter automata, their semantics, and give a first hierarchy of the one-counter languages considered so far in Section 4.3.2. Then, in Section 4.3.3, we introduce *behavior graphs*, which are infinite automata describing the behavior of a visibly one-counter automaton, and state their useful properties. Finally, the learning algorithm is given in Section 4.3.4.

4.3.1. Visibly one-counter automata

As said above, a visibly one-counter automaton is a DOCA where ε-transitions are forbidden and in which each counter operation is dictated by the read input symbol. For instance, each time we read an a in a run of a VOCA, we must increment the counter by one. We first introduce the notion of *pushdown alphabet*, which gives the operation to apply for each symbol.

Definition 4.3.1 (Pushdown alphabet)**.** A *pushdown alphabet*, denoted by $\widetilde{\Sigma} = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$, is the union of three disjoint alphabets:

- Σ_c is the set of *calls* where every call increments the counter by one,
- Σ_r is the set of *returns* where every return decrements the counter by one, and
- Σ_{int} is the set of *internal symbols* where an internal symbol does not change the counter.

The *sign of* a symbol $a \in \widetilde{\Sigma}$, denoted by $sign(a)$, is the counter operation it induces:

$$sign(a) = \begin{cases} 1 & \text{if } a \in \Sigma_c \\ -1 & \text{if } a \in \Sigma_r \\ 0 & \text{if } a \in \Sigma_{int}. \end{cases}$$

Since each symbol has a unique operation associated to it, it means that all automata using a given pushdown alphabet will necessarily have the same counter value for a word. That is, unlike for DOCAs, we can define the counter value of a word and its height solely from the alphabet.

Definition 4.3.2 (Counter value and height)**.** The *counter value* of a word $w = a_1 \cdots a_n \in \widetilde{\Sigma}^*$, denoted by $cv(w)$, is the sum of the signs of a_1 to a_n, *i. e.*,

$$cv(w) = \sum_{\ell=1}^{n} sign(a_\ell).$$

The *height* of w, denoted by $height(w)$, is the maximal counter value of any of its prefixes, *i. e.*,

$$height(w) = \max_{u \in Pref(w)} cv(u).$$

Observe that $cv(\varepsilon) = 0$. We highlight that the counter value of a word $u \cdot v$ can be obtained by summing the counter values of u and v, as stated in the next proposition. The proof follows immediately from the definition of a pushdown alphabet.

Proposition 4.3.3. *For two words u, v over a pushdown alphabet $\widetilde{\Sigma}$, we have*

$$cv(uv) = cv(u) + cv(v).$$

We can now define VOCAs. For clarity, we give here a full definition instead of restricting the definition of a DOCA. We highlight that in [NL10], one can test the counter value against any natural between 0 and m, with m a natural parameter of the VOCA, and check if it is greater than m. That is, the set of allowed guards is $\{=0, \dots, =m, >m\}$. For the sake of Chapter 5, we fix $m = 0$, *i. e.*, the guards we consider are $\{=0, >0\}$. This restricts the set of languages that can be described. As we did for DOCAs, we define *sound* VOCAs where the counter can never go below zero. That is, it is not possible to read a return symbol when the counter is already zero.

Definition 4.3.4 (Visibly one-counter automaton [NL10])**.** A *visibly one-counter automaton* (*VOCA*, for short) is a tuple $\mathcal{A} = (\widetilde{\Sigma}, Q, q_0, F, \delta)$ where:

- $\widetilde{\Sigma}$ is a pushdown alphabet,
- Q is a non-empty finite set of states, with $q_0 \in Q$ initial state,
- $F \subseteq Q$ is the set of final states, and
- $\delta : Q \times \widetilde{\Sigma} \times \{=0, >0\} \rightharpoonup Q$ is a deterministic transition function. As usual, we write $q \xrightarrow{a[g]} q'$ if $\delta(q, a, g) = q'$.

We say that $\mathcal{A}$ is *sound* if for every $q \xrightarrow{a[=0]} q'$ it holds that $a \notin \Sigma_r$.

As usual, we add a superscript to indicate which automaton is considered, and missing symbols are quantified existentially.

A *run* of $\mathcal{A}$ either consists of a single state p_0 or of a nonempty sequence of transitions

$$p_0 \xrightarrow{a_1[g_1]} p_1 \xrightarrow{a_2[g_2]} \cdots \xrightarrow{a_n[g_n]} p_n.$$

We denote by $runs(\mathcal{A})$ the set of runs of $\mathcal{A}$. Again, we often write $q \xrightarrow{a[g]} \in runs(\mathcal{A})$ to highlight that $\delta(q, a, g)$ is defined. Note that any run is uniquely determined by its first state and the sequence of symbols and guards.

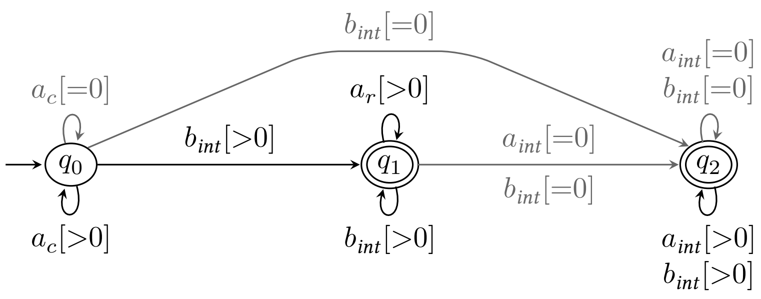


Figure 4.3: An example of a VCA.

Example 4.3.5. Let $\widetilde{\Sigma} = \{a_c\} \cup \{a_r\} \cup \{a_{int}, b_{int}\}$ be a pushdown alphabet. Then,

$$\begin{aligned} cv(a_c a_c b_{int} a_r b_{int} a_r b_{int}) &= 1 + 1 + 0 - 1 + 0 - 1 + 0 \\ &= 0 \end{aligned}$$

and

$$height(a_c a_c b_{int} a_r b_{int} a_r b_{int}) = cv(a_c a_c) = 2.$$

A 3-state VOCA $\mathcal{A}$ over the pushdown alphabet $\widetilde{\Sigma}$ is given in Figure 4.3. Transitions functions give the input symbol and the guard between square brackets. In order to help the reading, transitions using the guard $[=0]$ are drawn in gray. A sample run is

$$\begin{aligned} q_0 &\xrightarrow{a_c[=0]} q_0 \xrightarrow{a_c[>0]} q_0 \xrightarrow{b_{int}[>0]} q_1 \xrightarrow{a_r[>0]} q_1 \\ &\xrightarrow{b_{int}[>0]} q_1 \xrightarrow{a_r[>0]} q_1 \xrightarrow{a_{int}[=0]} q_2. \end{aligned}$$

It is not hard to see that $\mathcal{A}$ is sound.

Semantics

The semantics of a sound VOCA is defined as for sound DOCAs, except that the counter operation to apply is deduced from the symbol (and not retrieved from δ). That is, for two configurations $(q, n), (p, m)$, there is a transition $(q, n) \xrightarrow{a} (p, m)$ if and only if $m = n + sign(a)$ and

$$p = \begin{cases} \delta(q, a, =0) & \text{if } n = 0 \\ \delta(q, a, >0) & \text{if } n > 0. \end{cases}$$

We then obtain a (potentially infinite) transition system between configurations. Recall that it is impossible to decrement a counter equal to zero, when $\mathcal{A}$ is sound. Counted runs are defined exactly as for DOCAs and we still write $cruns(\mathcal{A})$ for the set of all counted runs of the sound VOCA $\mathcal{A}$.

Again, missing symbols in $(q, n) \xrightarrow{a} (p, m)$ are quantified existentially. We lift the notation to words as usual:

$$(p_0, n_0) \xrightarrow{a_1 \cdots a_\ell} (p_\ell, n_\ell) \in cruns(\mathcal{A})$$

if there exists a counted run

$$(p_0, n_0) \xrightarrow{a_1} \cdots \xrightarrow{a_\ell} (p_\ell, n_\ell) \in \mathit{cruns}(\mathcal{A}).$$

A counted run is uniquely determined by its first configuration and word.

Then, the language of a VOCA is easy to define to define: a word w is accepted by $\mathcal{A}$ if there is a counted run labeled by w from the *initial configuration* $(q_0, 0)$ to some configuration $(q, 0)$ with $q \in F$.

Definition 4.3.6 (Visibly one-counter language)**.** The language accepted by a sound VOCA $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{w \in \widetilde{\Sigma}^* \mid \exists q \in F : (q_0, 0) \xrightarrow{w} (q, 0) \in \mathit{cruns}(\mathcal{A})\}.$$

We say that a language L is a *visibly one-counter language* (VOCL, for short) if there is a sound VOCA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

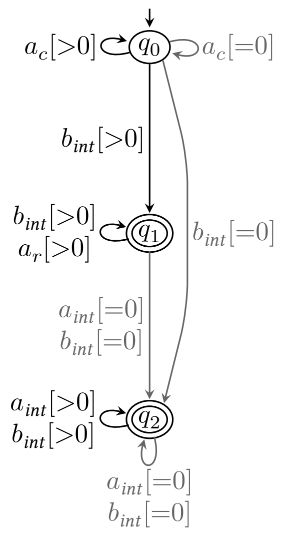
Example 4.3.7. Let $\mathcal{A}$ be the sound VOCA of Figure 4.3, which is repeated in the margin. A sample counted run is:

$$\begin{aligned}(q_0, 0) \xrightarrow{a_c} (q_0, 1) \xrightarrow{a_c} (q_0, 2) \xrightarrow{b_{\mathit{int}}} (q_1, 2) \xrightarrow{a_r} (q_1, 1) \\ \xrightarrow{b_{\mathit{int}}} (q_1, 1) \xrightarrow{a_r} (q_1, 0) \xrightarrow{a_{\mathit{int}}} (q_2, 0).\end{aligned}$$

Since the last configuration is $(q_2, 0)$ and $q_2 \in F$, $a_c \cdot a_c \cdot b_{\mathit{int}} \cdot a_r \cdot b_{\mathit{int}} \cdot a_r \cdot a_{\mathit{int}}$ is accepted by $\mathcal{A}$. By adding the appropriate guards (that depend on the current counter value) and dropping the counter values from the configurations, one can obtain the run from Example 4.3.5.
Now, let us consider the run

$$q_0 \xrightarrow{a_c[=0]} q_0 \xrightarrow{a_c[=0]} q_0 \xrightarrow{b_{\mathit{int}}[=0]} q_2.$$

We argue that for any starting counter value $n \geq 0$, it is impossible to construct a counted run that uses the same transitions of $\mathcal{A}$. Since the first symbol to process is $a_c \in \Sigma_c$, we thus obtain that the next counter value is $n + 1 > 0$. So, no matter the value of n, the second guard can never be satisfied.



4.3.2. A hierarchy of one-counter languages

Let us draw a hierarchy of the families of one-counter languages defined so far (including regular languages), which will be extended in Chapter 5. First, we show that the set of all VOCLs is a strict superset of the set of regular languages. Second, we argue that any VOCL is a DOCL but that there exists a DOCL for which there is no VOCA. Third, we prove that allowing non-determinism in one-counter automata increases the expressivity. Finally, we quickly state that one-counter languages are a strict subset of context-free languages (see [HU79] for an introduction to context-free languages). We do not provide proper proofs but focus instead on the intuitions behind them. Figure 4.4 gives a visual representation of the resulting hierarchy.

[HU79]: Hopcroft et al. (1979), *Introduction to Automata Theory, Languages and Computation*

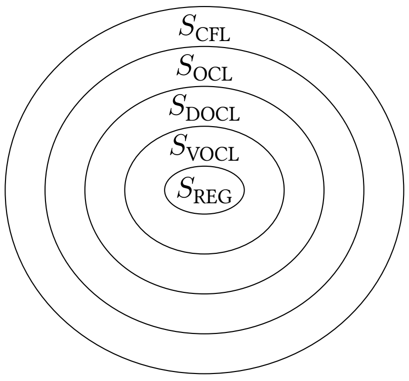


Figure 4.4: A visual representation of the hierarchy of one-counter languages. Each S_i denotes the set of all languages of type i. CFL designates the set of context-free languages, while OCL stands for (nondeterministic) one-counter languages.

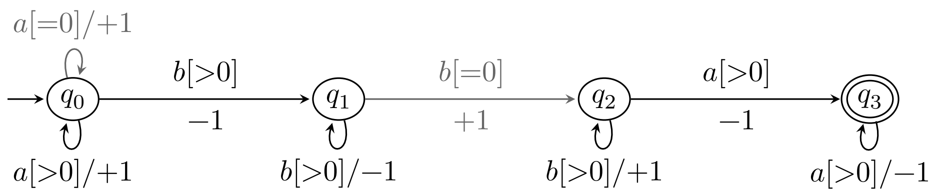


Figure 4.5: A DOCA accepting the language $\{a^n b^n b^m a^m \mid 0 < n, m\}$.

Proposition 4.3.8. *Any regular language is a VOCL but there exists a VOCL that is not regular.*

Sketch of proof. Let $L_{\text{REG}} \subseteq \Sigma$ be a regular language accepted by some DFA $\mathcal{A}$. Moreover, let $\widetilde{\Sigma}$ be a pushdown alphabet such that all symbols are internal, *i.e.*, $\Sigma_c = \Sigma_r = \emptyset$ and $\Sigma_{int} = \Sigma$. Then, construct a VOCA $\mathcal{B}$ over $\widetilde{\Sigma}$ by copying $\mathcal{A}$ and using only the guard $=0$. Since the counter is never modified (as all symbols are internal), it is indeed sufficient to only define transitions with $=0$. It is not hard to see that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B}) = L_{\text{REG}}$. Hence, any regular language is a VOCL.
Now, let $\widetilde{\Sigma}$ be a pushdown alphabet such that $\Sigma_c = \{a\}, \Sigma_r = \{b\}$, and $\Sigma_{int} = \emptyset$, and $L_{\text{VOCL}} = \{a^n b^n \mid n > 0\}$ be a language. It is well-known that there exists no DFA for L_{VOCL} (one can show it by a pumping argument. See [HU79], for instance). Hence, L_{VOCL} is not regular. □

[HU79]: Hopcroft et al. (1979), *Introduction to Automata Theory, Languages and Computation*

Proposition 4.3.9. *Any VOCL is a DOCL but there exists a DOCL that is not a VOCL.*

Sketch of proof. Given the definitions of VOCA and DOCA, it is clear that any VOCA can be transformed into a DOCA. Hence, every VOCL can be accepted by some DOCA and is thus a DOCL.
For the other direction, let $L_{\text{DOCL}} = \{a^n b^n b^m a^m \mid 0 < n, m\}$ be a language over the alphabet $\{a, b\}$. Figure 4.5 gives a DOCA accepting L_{DOCL}. Since the counter has to be incremented when reading the n first a and decremented it when reading the m last a, it is impossible to construct a pushdown alphabet. Indeed, a has to be a call and a return symbol at the same time, which is not

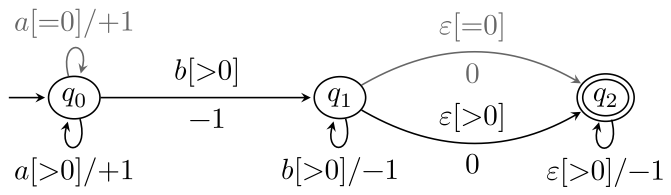


Figure 4.6: An OCA accepting the language $\{a^n b^m \mid 0 < m \leq n\}$.

permitted. Hence, no VOCA can accept L_{DOCL}. □

Let us quickly introduce (nondeterministic) *one-counter automaton* (*OCA*, for short). Unlike in a DOCA, it is permitted to have $q \xrightarrow{a[g]}$ and $q \xrightarrow{\varepsilon[g]}$ defined at the same time. Furthermore, the transitions are given as a relation (*i.e.*, there can be multiple pairs of target state and counter operation for a given tuple (q, a, g)). That is, an OCA is an NFA augmented with a counter and with ε-transitions. One can easily adapt the definition of counted runs and language to OCAs. A language L is a *one-counter language* (*OCL*, for short) if there exists an OCA accepting L.

Proposition 4.3.10. *Any DOCL is an OCL but there exists an OCL that is not a DOCL.*

Sketch of proof. Again, by definition, it is clear that any DOCA can be seen as an OCA. Hence, every DOCL is an OCL.
For the other direction, let us consider the language $L_{\text{OCL}} = \{a^n b^m \mid 0 < m \leq n\}$. Figure 4.6 gives an OCA accepting L_{OCL}.[4] We argue that there exists no DOCA accepting that language. Towards a contradiction, assume $\mathcal{A}$ is a DOCA such that $\mathcal{L}(\mathcal{A}) = L$. Let us fix some $n > |Q^{\mathcal{A}}|$. We then have the counted run

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{a^n} (q, c) \in \mathit{cruns}(\mathcal{A})$$

for some $q \in Q^{\mathcal{A}}$ and $c \in \mathbb{N}$. Since $a^n b^m \in L$ for every $0 < m \leq n$ and by the acceptance condition of a DOCA, it must be that

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{a^n} (q, c) \xrightarrow{b} (p_1, 0) \xrightarrow{b} \cdots \xrightarrow{b} (p_m, 0) \in \mathit{cruns}(\mathcal{A})$$

with $p_i \in F^{\mathcal{A}}$ for every $i \in \{1, \dots, m\}$. If we take $m > |Q^{\mathcal{A}}|$ (and such that $m \leq n$), there must exist $i < j \in \{1, \dots, m\}$ such that $p_i = p_j$. Moreover, we know that

$$(q, c) \xrightarrow{b^i} (p_i, 0) \xrightarrow{b^{j-i}} (p_i, 0) \in \mathit{cruns}(\mathcal{A}).$$

We can thus pump this loop reading b^{j-i} as many times as we want, say $n + 1$ times to obtain the counted run

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{a^n} (q, c) \xrightarrow{b^i} (p_i, 0) \xrightarrow{(b^{j-i})^{n+1}} (p_i, 0) \in \mathit{cruns}(\mathcal{A}).$$

As $p_i \in F^{\mathcal{A}}$, $a^n \cdot b^i \cdot b^{(j-i)(n+1)} \in \mathcal{L}(\mathcal{A})$. However, as $(j - i)(n + 1) > n$ (as $j - i \geq 1$), the word cannot belong to L. Hence, $\mathcal{L}(\mathcal{A}) \neq L$, which is a contradiction. We conclude that L cannot be a DOCL. □

4: We highlight that L_{OCL} is different from the language of Example 4.2.5, which is accepted by a DOCA.

Finally, we show that OCLs form a subset of *context-free languages* (*CFL*, for short), which are the languages accepted by pushdown automata. That is, we argue that a counter allows strictly less expressivity than a stack.

Proposition 4.3.11. *Any OCL is a CFL but there exists a CFL that is not an OCL.*

Proof. Since one-counter automata can be seen as pushdown automata with a single stack symbol, any OCL is a CFL.
On the opposite, let $L_{\text{CFL}} = \{w \cdot w^R \mid w \in \{0,1\}^*\}$, where w^R denotes the reverse word of w (*e.g.*, $(abc)^R = cba$). It is not hard to construct a pushdown automaton accepting L_{CFL} (see [HU79]). While we do not do it here, one can show that a counter is not sufficient to describe the reverse word of w when the alphabet has strictly more than one symbol. □

[HU79]: Hopcroft et al. (1979), *Introduction to Automata Theory, Languages and Computation*

4.3.3. Behavior graph

Let us now introduce a finite representation for the (infinite) transition system of VOCAs, as proposed by Neider and Löding in [NL10] and used in their learning algorithm. The idea is to rely on the Myhill-Nerode congruence (see Definition 2.2.6) to define a deterministic infinite automaton. This automaton is shown to always have an ultimately periodic structure. Hence, it is sufficient to encode the periodic behavior and the initial fragment up to it, which can be done in finite memory.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

Definition 2.2.6. For a language $L \subseteq \Sigma^*$ and two words $u, v \in \Sigma^*$, we write $u \sim_L v$ if for all $w \in \Sigma^*$, we have $uw \in L \Leftrightarrow vw \in L$.

Assumption 4.3.12. In order to avoid having to treat particular cases, we assume from now on that any considered VOCL is non-empty.

First, we prove that two equivalent words according to the Myhill-Nerode congruence $\sim_L$ have the same counter value, if they are in the prefix of the language L.

Lemma 4.3.13. *Let $L \subseteq \widetilde{\Sigma}^*$ be a VOCL and $u, v \in \mathit{Pref}(L)$ such that $u \sim_L v$. Then, $\mathit{cv}(u) = \mathit{cv}(v)$.*

Proof. Let $u, v \in \mathit{Pref}(L)$. Since $u \in \mathit{Pref}(L)$, there exists $w \in \widetilde{\Sigma}^*$ such that $uw \in L$ and $vw \in L$ (as $u \sim_L v$). We have $\mathit{cv}(uw) = \mathit{cv}(u) + \mathit{cv}(w) = 0$ and $\mathit{cv}(vw) = \mathit{cv}(v) + \mathit{cv}(w) = 0$. We conclude that $\mathit{cv}(u) = \mathit{cv}(v)$. □

Definition 2.2.9. Let $L \subseteq \Sigma^*$ be a language. We define the automaton $\mathcal{A}_{\sim_L} = (\Sigma, Q, q_0, F, \delta)$ with

- $Q = \{[\![w]\!]_{\sim_L} \mid w \in \Sigma^*\}$,
- $q_0 = [\![\varepsilon]\!]_{\sim_L}$,
- $F = \{[\![w]\!]_{\sim_L} \mid w \in L\}$, and
- $\delta : Q \times \Sigma \to Q$ is the (total) function defined such that for all $w \in \Sigma^*$ and $a \in \Sigma$,

$$[\![w]\!]_{\sim_L} \xrightarrow{a} [\![w \cdot a]\!]_{\sim_L} \in \mathit{runs}(\mathcal{A}_{\sim_L}).$$

This allows us to define a (potentially infinite) deterministic automaton from $\sim_L$, called the *behavior graph* of the language L. We gave in Definition 2.2.9 the definition of an automaton from $\sim_L$, using every equivalence class of $\sim_L$. Here, we restrict the set of states to the classes $[\![w]\!]_{\sim_L}$ where w is in the set of prefixes of L. That is, the classes for words that are not in the prefix are of no interest to us. In other words, we only keep the states that are reachable from the initial state and co-reachable from some final state. Hence, the transition function is partial.[5] Observe that there necessarily exists a reachable and co-reachable equivalence class by Assumption 4.3.12, *i.e.*, the automaton is not empty.

5: The definition of behavior graph from [NL10] is different as non-co-reachable states are allowed. However, the two coincide on reachable and co-reachable states.

Definition 4.3.14 (Behavior graph [NL10])**.** Let $L \subseteq \widetilde{\Sigma}^*$ be a VOCL, $\sim_L$ be its Myhill-Nerode congruence, and $\mathcal{A}$ be the deterministic (in)finite automaton defined from $\sim_L$. The *behavior graph* of L is the tuple

$$BG(L) = (\widetilde{\Sigma}, Q^{BG(L)}, q_0^{BG(L)}, F^{BG(L)}, \delta^{BG(L)})$$

obtained from $\mathcal{A}$ by restricting the set of states to

$$Q^{BG(L)} = \{[\![w]\!]_{\sim_L} \mid w \in \mathit{Pref}(L)\}.$$

It holds that any class $[\![w]\!]_{\sim_L} \in F^{BG(L)}$ is such that $\mathit{cv}(w) = 0$, by definition of a VOCL. Let us quickly argue that $BG(L)$ accepts L.

Proposition 4.3.15 ([NL10])**.** *For any VOCL L, $\mathcal{L}(BG(L)) = L$.*

Sketch of proof. As L is a VOCL, there exists a VOCA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$. One can show by a straight-forward induction that

$$\forall (q_0^{\mathcal{A}}, 0) \xrightarrow{w} (q, n) \in \mathit{cruns}(\mathcal{A}) : q_0^{BG(L)} \xrightarrow{w} [\![w]\!]_{\sim_L}.$$

The other direction also holds.
By definition of $BG(L)$ from $\sim_L$, it is clear that

$$w \in L \Leftrightarrow [\![w]\!]_{\sim_L} \in F^{BG(L)}.$$

Moreover, since $\mathcal{L}(\mathcal{A}) = L$,

$$\begin{aligned} w \in L &\Leftrightarrow w \in \mathcal{L}(\mathcal{A}) \\ &\Leftrightarrow \exists q \in F^{\mathcal{A}} : (q_0^{\mathcal{A}}, 0) \xrightarrow{w} (q, 0) \\ &\Leftrightarrow q_0^{BG(L)} \xrightarrow{w} [\![w]\!]_{\sim_L}. \end{aligned}$$

Hence,

$$w \in L \Leftrightarrow w \in \mathcal{L}(\mathcal{A}) \Leftrightarrow w \in \mathcal{L}(BG(L)).$$ □

By Lemma 4.3.13, we know that all words in an equivalence class have the same counter value. Hence, we can group together all the states of $BG(L)$ that share the same counter value ℓ, called the *level* ℓ. The *width* of $BG(L)$ is then defined as the maximal number of classes among all levels.

Definition 4.3.16 (Level and width)**.** The *level* $\ell \in \mathbb{N}$ of $BG(L)$, denoted by $\mathit{level}(BG(L), \ell)$ is the set of states with counter value ℓ, *i.e.*,

$$\mathit{level}(BG(L), \ell) = \{[\![w]\!]_{\sim_L} \in Q^{BG(L)} \mid \mathit{cv}(w) = \ell\}.$$

The *width* of $BG(L)$, denoted by $\mathit{width}(BG(L))$, is the maximal size of any level:

$$\mathit{width}(BG(L)) = \max_{\ell \in \mathbb{N}} |\mathit{level}(BG(L), \ell)|.$$

It is shown in [NL10] that the number of states in each level is bounded by a constant $K \in \mathbb{N}$, *i.e.*, the width is bounded. In particular, one can take K equal to the number of states of any VOCA accepting L.

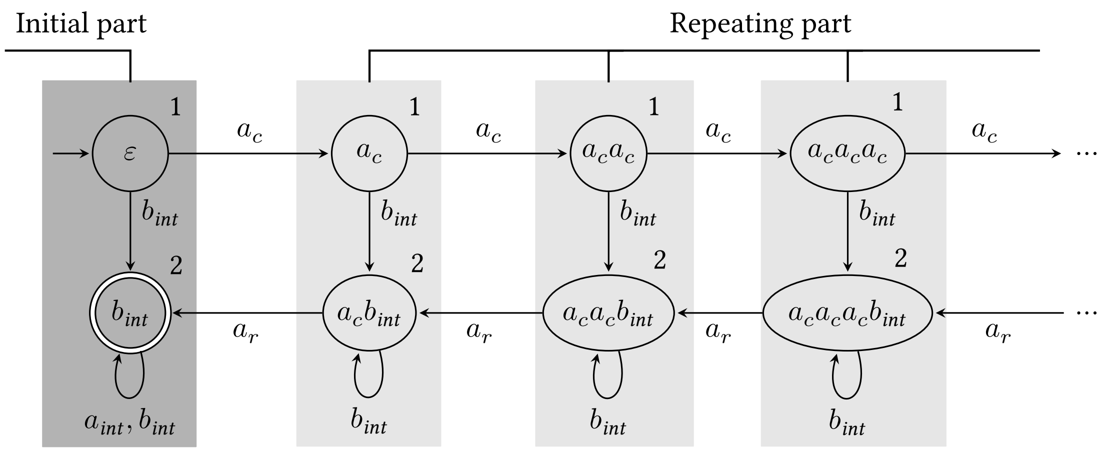
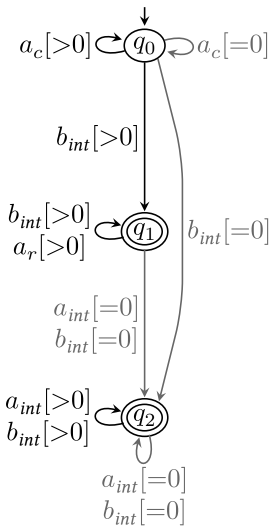


Figure 4.7: Behavior graph of the language accepted by the VOCA from Figure 4.3.



Lemma 4.3.17 ([NL10]). *Let $\mathcal{A}$ be a VOCA accepting a VOCL L, and $BG(L)$ be the behavior graph of L. Then, $\mathit{width}(BG(L)) \leq |Q^{\mathcal{A}}|$.*

Example 4.3.18. Let L be the language accepted by the VCA $\mathcal{A}$ of Figure 4.3, which is repeated in the margin. Its behavior graph $BG(L)$ is given in Figure 4.7. For clarity sake, each state $[\![w]\!]_{\sim_L}$ is represented by one of its words, and all states in the same level are vertically aligned. For instance, $\mathit{level}(BG(L), 0)$ is comprised of the classes of ε and b_{int}, as $\mathit{cv}(\varepsilon) = \mathit{cv}(b_{\mathit{int}}) = 0$. We can see that $\mathit{width}(BG(L)) = 2$.

Finite representation

Neider and Löding proved that $BG(L)$ always has a finite representation, even though the graph itself is potentially infinite. This finite representation relies on the fact that $BG(L)$ has an ultimately periodic structure, *i.e.*, it has an "initial part" that is followed by a "repeating part" repeated ad infinitum. We give here a short overview of the idea and refer to [NL10] for further details.

Let K be the width of $BG(L)$. The fact that K is bounded (see Lemma 4.3.17) allows us to enumerate the states at level ℓ via a mapping

$$\nu_\ell : \mathit{level}(BG(L), \ell) \to \{1, \dots, K\}.$$

Using these enumerations ν_ℓ, $\ell \in \mathbb{N}$, we can encode the transitions of $BG(L)$ as a sequence of (partial) mappings

$$\tau_\ell : \{1, \dots, K\} \times \widetilde{\Sigma} \rightharpoonup \{1, \dots, K\}.$$

For all $\ell \in \mathbb{N}, i \in \{1, \dots, K\}$, and $a \in \widetilde{\Sigma}$, the mapping τ_ℓ is defined as $\tau_\ell(i, a) = j$ if there exist $[\![u]\!]_{\sim_L}, [\![ua]\!]_{\sim_L} \in Q^{BG(L)}$ such that

- $\mathit{cv}(u) = \ell$,
- $\nu_\ell([\![u]\!]_{\sim_L}) = i$, and
- $\nu_{\ell+\mathit{sign}(a)}([\![ua]\!]_{\sim_L}) = j$,

and is left undefined otherwise.

Thus, if we fix the enumerations ν_ℓ, the behavior graph can be encoded as the sequence of mappings $\alpha = \tau_0\tau_1\tau_2 \dots$, called a *description* of $BG(L)$. The following theorem states that there always exists such a description which is ultimately periodic.

Theorem 4.3.19 ([NL10]). *Let $BG(L)$ be the behavior graph of a VOCL L. Then, there exists, for each level ℓ, an enumeration*

$$\nu_\ell : \mathit{level}(BG(L), \ell) \to \{1, \dots, \mathit{width}(BG(L))\},$$

such that the corresponding description α of $BG(L)$ is an ultimately periodic word with offset $m > 0$ and period $k \geq 0$, i. e.,

$$\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega.$$

Conversely, from an ultimately periodic description of $BG(L)$, it is possible to construct a VOCA accepting L [NL10, Lemma 1].

Example 4.3.20. Let us continue Example 4.3.18. We explain why $BG(L)$ has a finite representation. First, we define the enumerations $\nu_\ell, \ell \in \mathbb{N}$:

$$\begin{aligned}
\nu_0(\llbracket\varepsilon\rrbracket_{\sim_L}) &= 1 & \nu_0(\llbracket b_{\mathit{int}}\rrbracket_{\sim_L}) &= 2\\
\nu_1(\llbracket a_c\rrbracket_{\sim_L}) &= 1 & \nu_1(\llbracket a_c b_{\mathit{int}}\rrbracket_{\sim_L}) &= 2\\
\nu_2(\llbracket a_c a_c\rrbracket_{\sim_L}) &= 1 & \nu_2(\llbracket a_c a_c b_{\mathit{int}}\rrbracket_{\sim_L}) &= 2\\
&\vdots & &\vdots
\end{aligned}$$

In Figure 4.7, each state has its associated number next to it.
From these enumerations, we define the following τ_ℓ mappings (missing values are undefined):

$$\begin{array}{cccc}
\tau_0(1, a_c) = 1 & \tau_0(1, b_{\mathit{int}}) = 2 & \tau_0(2, a_{\mathit{int}}) = 2 & \tau_0(2, b_{\mathit{int}}) = 2\\
\tau_1(1, a_c) = 1 & \tau_1(1, b_{\mathit{int}}) = 2 & \tau_1(2, a_r) = 2 & \tau_1(2, b_{\mathit{int}}) = 2\\
\tau_2(1, a_c) = 1 & \tau_2(1, b_{\mathit{int}}) = 2 & \tau_2(2, a_r) = 2 & \tau_2(2, b_{\mathit{int}}) = 2\\
\vdots & \vdots & \vdots & \vdots
\end{array}$$

We can thus define the description $\alpha = \tau_0\tau_1\tau_2 \dots$ of $BG(L)$. As $\tau_1 = \tau_2 = \tau_3 = \cdots$, we have that $\alpha = \tau_0(\tau_1)^\omega$, *i. e.*, α is an ultimately periodic description of $BG(L)$.

4.3.4. Learning visibly one-counter languages

We now focus on L^*_{VOCA}, the L^* adaptation for VOCAs introduced in [NL10]. We refer to Section 3.2 for an introduction to L^* for DFAs. Let L be some VOCL. The idea is to learn a fragment of the behavior graph of L up to a fixed counter limit ℓ, extract every possible ultimately periodic description of the fragment, and construct a VOCA from each of these descriptions. If we find one VOCA accepting L, we are done. Otherwise, we increase ℓ and repeat the process.

We first properly define the fragment of the behavior graph up to ℓ and explain how to extract (ultimately periodic) descriptions from it, from which VOCAs can be constructed. Once it is done, we can fix the learning framework (*i. e.*, define the queries the learner can use). Finally, we introduce the algorithm itself and state its complexity.

Bounded behavior graph

Formally, the fragment of $BG(L)$ up to ℓ is a subgraph of $BG(L)$ such that the counter value of the states of that subgraph never exceeds ℓ. That is, the *height* of the words that can be read within that subgraph never exceeds ℓ.

Definition 4.3.21 (Bounded behavior graph)**.** Let $BG(L)$ be a behavior graph for some VOCL L. The *bounded behavior graph up to ℓ* is the DFA $BG_{\leq \ell}(L)$ such that

$$Q^{BG_{\leq \ell}(L)} = \{[\![w]\!]_{\sim_L} \in Q^{BG(L)} \mid \mathit{height}(w) \leq \ell\}.$$

The values of the (partial) function $\delta^{BG_{\leq \ell}(L)}$, the initial state, and the final states are naturally defined over the subgraph.
The language of $BG_{\leq \ell}(L)$ is denoted by $L_{\leq \ell}$.

Observe that $L_{\leq \ell}$ is regular, as $BG_{\leq \ell}(L)$ is a DFA.

We now explain how to construct VOCAs from $BG_{\leq \ell}(L)$. To do so, we extract every possible ultimately periodic description of the bounded behavior graph, by identifying an isomorphism between two consecutive subgraphs of $BG_{\leq \ell}(L)$. That is, we fix values for the offset m and period k and see if the subgraphs induced by the levels m to $m + k - 1$, and by the levels $m + k$ to $m + 2k - 1$ are isomorphic. Note that this means we need to consider all pairs of m and k such that $m + 2k - 1 \leq \ell$. This can be done in polynomial time by executing two depth-first searches in parallel [NL10]. Note also that multiple ultimately periodic descriptions may be found, due to the finite knowledge of the learner. From each ultimately periodic description, a VOCA can be built, *i. e.*, the procedure potentially yields many different VOCAs.

Queries

During learning, the learner can use membership queries and check whether a hypothesis VOCA accepts the target language L. As the learning algorithm will construct bounded behavior graphs up to some natural ℓ (that will increase each iteration), we must ensure that $L_{\leq \ell}$ is correctly learned. In general, an antagonistic teacher may answer increasingly longer counterexamples to VOCA equivalence queries, meaning that we can never conclude that $L_{\leq \ell}$ is correctly learned. Hence, we require a new query that explicitly checks whether a DFA accepts $L_{\leq \ell}$. Figure 4.8 gives a visual representation of the adapted Angluin's framework.

Definition 4.3.22 (Queries for VOCLs)**.** Let L be the VOCL of the teacher. A learner for VOCAs can use three types of queries:

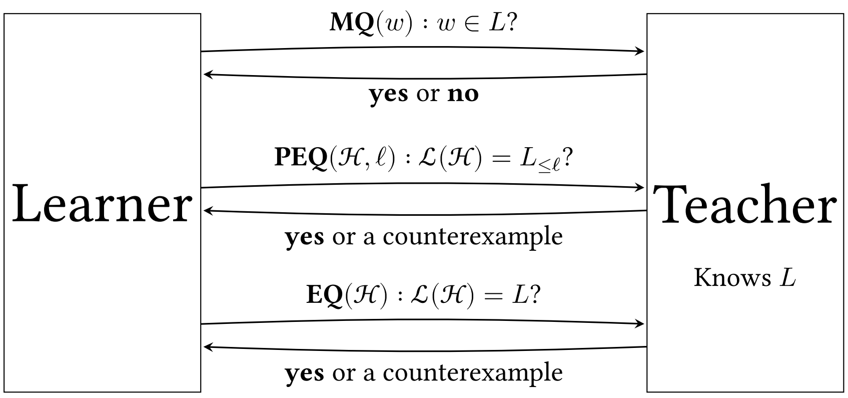

$\mathbf{MQ}(w) : w \in L?$
yes or no
$\mathbf{PEQ}(\mathcal{H}, \ell) : \mathcal{L}(\mathcal{H}) = L_{\leq \ell}?$
yes or a counterexample
$\mathbf{EQ}(\mathcal{H}) : \mathcal{L}(\mathcal{H}) = L?$
yes or a counterexample
Learner
Teacher
Knows L

Figure 4.8: Adaptation of Angluin's framework for VOCAs.

- A *membership query*, denoted by $\mathbf{MQ}(w)$, with $w \in \widetilde{\Sigma}^*$, returns whether $w \in L$.
- A *partial equivalence query*, denoted by $\mathbf{PEQ}(\mathcal{H}, \ell)$, with $\mathcal{H}$ a DFA over $\widetilde{\Sigma}$ and $\ell \in \mathbb{N}$, returns
  - **yes** if $\mathcal{L}(\mathcal{H}) = L_{\leq \ell}$
  - or a word w such that $w \in \mathcal{L}(\mathcal{H}) \Leftrightarrow w \notin L_{\leq \ell}$
- An *equivalence query*, denoted by $\mathbf{EQ}(\mathcal{H})$, with $\mathcal{H}$ a VOCA over $\widetilde{\Sigma}$, returns
  - **yes** if $\mathcal{L}(\mathcal{H}) = L$,
  - or a word w such that $w \in \mathcal{L}(\mathcal{H}) \Leftrightarrow w \notin L$.

Learning algorithm

The data structure of the learner is an observation table, as in L^*, except that a counter-value limit ℓ is taken into account.[6] The aim of the table is to approximate the equivalence classes of $\sim_{L_{\leq \ell}}$ to learn the bounded behavior graph $BG_{\leq \ell}(L)$. As said above, ℓ will increase over time.

6: In [NL10], a so-called *stratified* observation table, composed of multiple observation tables (one per level of the behavior graph) is used. Here, we present a single table.

Definition 4.3.23 (Observation table for VOCAs). Let $\ell \in \mathbb{N}$ be a counter-value limit. An *observation table up to* ℓ is a tuple $\mathcal{O}_{\leq \ell} = (R, S, T)$ with

- $R \subsetneq \widetilde{\Sigma}^*$ a finite prefix-closed set of representatives,
- $S \subsetneq \widetilde{\Sigma}^*$ a finite suffix-closed set of separators,
- $T : (R \cup R\widetilde{\Sigma}) \cdot S \to \{\mathbf{no}, \mathbf{yes}\}$ such that

$$\forall w \in (R \cup R\widetilde{\Sigma}) \cdot S : T(w) = \mathbf{yes} \Leftrightarrow w \in L_{\leq \ell}.$$

Since $L_{\leq \ell}$ is regular, one can learn it by applying the classical L^* algorithm. That is, we fix a maximal counter value ℓ and refine $\mathcal{O}_{\leq \ell}$ until it is closed and Σ-consistent. From the resulting observation tree, a hypothesis DFA $\mathcal{H}$ can be constructed. If $\mathcal{H}$ accepts $L_{\leq \ell}$, then one can extract ultimately periodic descriptions and VOCAs, as explained above.

Recall that multiple VOCAs may be yielded. We ask an equivalence query for each of them. If one of the VOCAs accepts the target language, we are

Algorithm 4.1: Overall L^*_{VOCA} algorithm.

```
1:  Initialize $\mathcal{O}_{\leq \ell}$ with $\ell = 0, R = S = \{\varepsilon\}$
2:  while true do
3:      Make $\mathcal{O}_{\leq \ell}$ closed and $\Sigma$-consistent                ▷ Using $L^*$
4:      Construct the DFA $\mathcal{H}_{\leq \ell}$ from $\mathcal{O}_{\leq \ell}$
5:      $v \leftarrow$ PEQ($\mathcal{H}_{\leq \ell}, \ell$)
6:      if $v \neq$ yes then
7:          Update $\mathcal{O}_{\leq \ell}$ with $v$                             ▷ $\ell$ is not modified
8:      else
9:          $W \leftarrow \emptyset$
10:         for all ultimately periodic descriptions $\alpha$ of $\mathcal{H}_{\leq \ell}$ do
11:             Construct the VOCA $\mathcal{H}_\alpha$ from $\alpha$
12:             $v_\alpha \leftarrow$ EQ($\mathcal{H}_\alpha$)
13:             if $v_\alpha =$ yes then return $\mathcal{H}_\alpha$
14:             else if $height(v_\alpha) > \ell$ then $W \leftarrow W \cup \{v_\alpha\}$
15:         if $W = \emptyset$ then
16:             Let $\mathcal{H}^{\mathrm{VOCA}}_{\leq \ell}$ be $\mathcal{H}_{\leq \ell}$ seen as a VOCA
17:             $w \leftarrow$ EQ($\mathcal{H}^{\mathrm{VOCA}}_{\leq \ell}$)
18:             if $w =$ yes then return $\mathcal{H}^{\mathrm{VOCA}}_{\leq \ell}$        ▷ The target language is regular
19:             else $W \leftarrow W \cup \{w\}$
20:         Select an arbitrary $w$ from $W$
21:         $\ell \leftarrow height(w)$                                          ▷ $\ell$ is increased
22:         Update $\mathcal{O}_{\leq \ell}$ with $w$
```

done. Otherwise, we need to refine the table and increase the counter limit ℓ. However, not all of the counterexamples given by the teacher can be used. Indeed, it may happen that the teacher returns a word that was incorrectly accepted or rejected by a VOCA, but for which the information is already present in the table (due to a description that covered only the first levels in $BG_{\leq \ell}(L)$, for instance). Therefore, we only consider counterexamples w with a height $height(w) > \ell$. If we cannot find such a counterexample, we use $BG_{\leq \ell}(L)$ directly as a VCA (where only the guard $=0$ is used). Since we know that $\mathcal{L}(BG_{\leq \ell}(L)) = L_{\leq \ell}$, we are guaranteed to obtain a useful counterexample.

Algorithm 4.1 gives a pseudo-code for L^*_{VOCA} while the next theorem summarizes its complexity.

Theorem 4.3.24 ([NL10]). *The L^*_{VOCA} algorithm has polynomial time and space complexity in the width of the behavior graph, the offset and period of an ultimately periodic description of the behavior graph, and the length of the longest counterexample returned on (partial) equivalence queries.*

Learning Realtime One-Counter Automata 5.

In this chapter, based on [BPS22], we present a learning algorithm for *realtime one-counter automata*[1], which are DOCAs where ε-transitions are forbidden. Our algorithm is based on L^*_{VOCA} [NL10] (presented in the previous chapter) and uses partial equivalence query and counter value queries, on top of the usual membership and equivalence queries. That is, we make the assumption that we have an executable black box with observable counter values. We prove that our algorithm runs in exponential time and space and that it uses at most an exponential number of queries. We also present and discuss experimental results obtained by learning randomly generated realtime one-counter automata, and a use case on constructing automata to validate JSON documents. We recommend reading the previous chapter before this one (in particular, we require ideas introduced in Section 4.3). Technical proofs and details are deferred to Appendix A.

[BPS22]: Bruyère et al. (2022), "Learning Realtime One-Counter Automata"

1: The term "realtime" denotes that the automaton takes exactly n computations steps to decide whether a word of length n is accepted.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

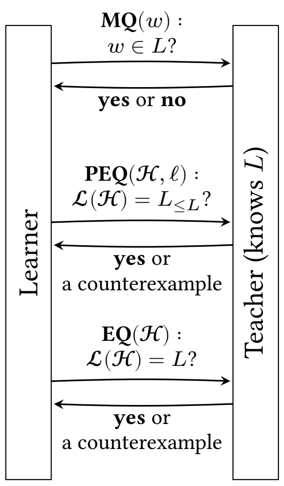
Chapter contents

5.1. Introduction

One-counter automata with counter-value observability were observed to have desirable properties by Bollig in [Bol16]. Importantly, in the same paper Bollig

[Bol16]: Bollig (2016), "One-Counter Automata with Counter Observability" highlights a connection between such automata and VOCAs. We expose a similar connection and are thus able to leverage Neider and Löding's L^*_{VOCA} algorithm [NL10] (see the previous chapter) as a sort of sub-routine for ours.

In the family of one-counter automata we consider in this chapter, called *realtime one-counter automata*, the counter values cannot be inferred from a given word anymore. Hence, we need *counter value queries* on top of the queries used by L^*_{VOCA}, which we recall in the margin. Moreover, we have to extend the classical definition of *observation tables* as used in, *e.g.*, [Ang87; NL10] (see Sections 3.2 and 4.3.4). Namely, entries in our tables are composed of Boolean language information as well as a counter value or a *wildcard* encoding the fact that we do not (yet) care about the value of the corresponding word. (Our use of wildcards is reminiscent of the work of Leucker and Neider [LN12] on learning a regular language from an "inexperienced" teacher who may answer queries in an unreliable manner.) Due to these extensions, much work is required to prove that it is always possible to make a table closed and consistent in finite time. A crucial element of our algorithm is that we formulate queries for the teacher in a way which ensures the observation table eventually induces a right congruence refining the classical Myhill-Nerode congruence with counter-value information. (This is in contrast with [LN12], where the ambiguity introduced by wildcards is resolved using SAT solvers.)



Learner
Teacher (knows L)
$\mathbf{MQ}(w)$: $w \in L$?
yes or **no**
$\mathbf{PEQ}(\mathcal{H}, \ell)$: $\mathcal{L}(\mathcal{H}) = L_{\leq L}$?
yes or a counterexample
$\mathbf{EQ}(\mathcal{H})$: $\mathcal{L}(\mathcal{H}) = L$?
yes or a counterexample

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

[LN12]: Leucker et al. (2012), "Learning Minimal Deterministic Automata from Inexperienced Teachers"

[CR04]: Chitic et al. (2004), "On Validation of XML Streams Using Finite State Machines"

[FR95]: Fahmy et al. (1995), "Efficient Learning of Real Time One-Counter Automata"

[BGJ14]: Böhm et al. (2014), "Bisimulation equivalence and regularity for real-time one-counter automata"

[BR87]: Berman et al. (1987), "Learning One-Counter Languages in Polynomial Time (Extended Abstract)"

[Roo88]: Roos (1988), *Deciding equivalence of deterministic one-counter automata in polynomial time with applications to learning*

We evaluate an implementation of our algorithm on random benchmarks and a use case inspired by [CR04]. Namely, we learn a realtime one-counter automaton model for a simple JSON schema validator — *i.e.*, a program that verifies whether a JSON document satisfies a given JSON schema. This idea will be further explored in Part III with a different family of automata.

It is noteworthy that, in [FR95], Fahmy and Roos claim to provide a learning algorithm for realtime one-counter automata. However, we were unable to understand the algorithm and proofs in that paper due to lack of precise formalization and detailed proofs. We also found an example where the provided algorithm did not produce the expected results. It is noteworthy that Böhm *et al.* [BGJ14] made similar remarks about related works of Roos [BR87; Roo88].

This chapter is structured as follows. We first define realtime one-counter automata and their semantics, extend the hierarchy initiated in Section 4.3.2, and introduce the learning framework (*i.e.*, the available queries) we consider. Then, in Section 5.3, we adapt the notion of behavior graphs introduced in the previous chapter. Our learning algorithm L^*_{ROCA} is explained in Section 5.4 where we also give its complexity. Finally, an experimental evaluation is performed in Section 5.5 before concluding in Section 5.6. Technical proofs and details are deferred to Appendix A.

5.2. Realtime one-counter automata

A *realtime one-counter automaton* is, in short, a DOCA where ε-transitions are forbidden. That is, it is a DFA augmented with a single natural counter where each transition can increment, decrement, or not modify that counter. For convenience, we give here a complete definition (instead of stating the

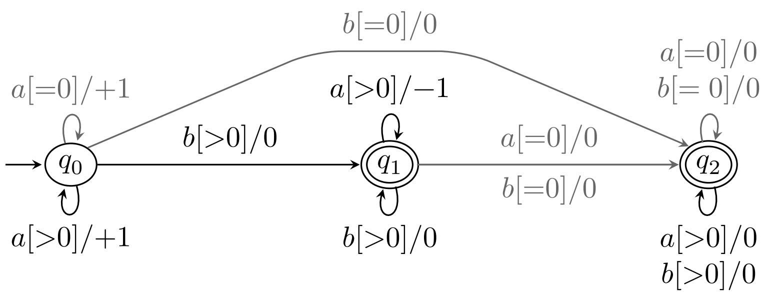


Figure 5.1: An example of an ROCA.

restrictions over a DOCA). We allow two types of *guards* on the transitions: $=0$ and >0, *i.e.*, we can check whether the counter is zero during an execution of the automaton. We also define *sound* ROCA in which the counter can never go below zero. That is, a transition cannot decrement the counter when the guard is $=0$. The model we present here is similar to the one from [FR95; VP75].

[FR95]: Fahmy et al. (1995), "Efficient Learning of Real Time One-Counter Automata"
[VP75]: Valiant et al. (1975), "Deterministic One-Counter Automata"

Definition 5.2.1 (Realtime one-counter automaton)**.** A *realtime one-counter automaton* (*ROCA*, for short) $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where:

- Σ is an alphabet,
- Q is a non-empty finite set of states, with $q_0 \in Q$ the initial state,
- $F \subseteq Q$ is the set of final states, and
- $\delta : Q \times \Sigma \times \{=0, >0\} \rightharpoonup Q \times \{-1, 0, +1\}$ is a deterministic (partial) transition function. As usual, we write $q \xrightarrow[c]{a[g]} q'$ if $\delta(q, a, g) = (q', c)$.

An ROCA $\mathcal{A}$ is termed *sound* if for every $q \xrightarrow[c]{a[=0]}$ it holds that $c \neq -1$.

As for DFAs, VOCAs, and so on, we add a superscript to indicate which automaton is considered, and missing symbols are quantified existentially.

A *run* of $\mathcal{A}$ either consists of a single state p_0 or of a nonempty sequence of transitions

$$p_0 \xrightarrow[c_1]{a_1[g_1]} p_1 \xrightarrow[c_2]{a_2[g_2]} \cdots \xrightarrow[c_n]{a_n[g_n]} p_n.$$

We denote by *runs*$(\mathcal{A})$ the set of runs of $\mathcal{A}$. As for finite automata, we often write $q \xrightarrow{a[g]} \in$ *runs*$(\mathcal{A})$ to highlight that $\delta(q, a, g)$ is defined. Note that any run is uniquely determined by its first state and the sequence of symbols and guards.

Example 5.2.2. A 3-state ROCA $\mathcal{A}$ over $\Sigma = \{a, b\}$ is given in Figure 5.1. The initial state q_0 is marked by a small arrow and the two final states q_1 and q_2 are double-circled. The transitions give the input symbol, the condition on the counter value, and the counter operation, in this order. In order to help the reading, transitions using the guard $[=0]$ are drawn in gray. A sample run is

$$q_0 \xrightarrow[+1]{a[=0]} q_0 \xrightarrow[+1]{a[>0]} q_0 \xrightarrow[0]{b[>0]} q_1 \xrightarrow[-1]{a[>0]} q_1$$
$$\xrightarrow[0]{b[>0]} q_1 \xrightarrow[-1]{a[>0]} q_1 \xrightarrow[0]{a[=0]} q_2.$$

It is not hard to see that $\mathcal{A}$ is sound.

5.2.1. Semantics

Let us now define the semantics of a sound ROCA $\mathcal{A}$ (in the same vein as what we did for DOCAs and VOCAs). We keep track of the current state q and the current counter value n in a *configuration*. If n is zero, we then process the next symbol a by retrieving the pair $(p, c) = \delta(q, a, {=}0)$, and applying the counter operation c on n. Likewise when $n > 0$, except that we pass ${>}0$ to δ. We then reach a new configuration $(p, n + c)$ from which we may process a new symbol, and so on. That is, the semantics of $\mathcal{A}$ are defined via a (potentially infinite) transition system. Recall that it is impossible to decrement a counter equal to zero, when $\mathcal{A}$ is sound.

Definition 5.2.3 (Counted runs)**.** Let $\mathcal{A}$ be a sound ROCA, $(q, n), (p, m) \in Q \times \mathbb{N}$ be two configurations, and $a \in \Sigma$ be a symbol. There exists a transition $(q, n) \xrightarrow{a} (p, m)$ if and only if $m = n + c$ and

$$(p, c) = \begin{cases} \delta(q, a, {=}0) & \text{if } n = 0 \\ \delta(q, a, {>}0) & \text{if } n > 0. \end{cases}$$

A *counted run* of $\mathcal{A}$ is either a configuration (p_0, n_0) or a nonempty sequence of transitions

$$(p_0, n_0) \xrightarrow{a_1} (p_1, n_1) \xrightarrow{a_2} \cdots \xrightarrow{a_\ell} (p_\ell, n_\ell).$$

We denote by *cruns*$(\mathcal{A})$ the set of all counted runs of $\mathcal{A}$.

Again, missing symbols in $(q, n) \xrightarrow{a} (p, m)$ are quantified existentially. We lift the notation to words as usual: $(p_0, n_0) \xrightarrow{a_1 \cdots a_\ell} (p_\ell, n_\ell) \in \mathit{cruns}(\mathcal{A})$ if there exists a counted run $(p_0, n_0) \xrightarrow{a_1} \cdots \xrightarrow{a_\ell} (p_\ell, n_\ell) \in \mathit{cruns}(\mathcal{A})$. A counted run is uniquely determined by its first configuration and word.

As for VOCAs, one can construct a run from a counted run. The other way does not hold in general (due to the counter values).

Let us adapt the notion of counter value and height (see Definition 4.3.2) to ROCAs. Since we no longer have a pushdown alphabet, these values must be determined from the ROCA itself. That is, different ROCAs over the same alphabet may have different counter values for a given word, unlike for VOCAs.

Definition 4.3.2. The *counter value* of a word $w = a_1 \cdots a_n \in \widetilde{\Sigma}^*$, denoted by $cv(w)$, is the sum of the signs of a_1 to a_n, *i.e.*, $cv(w) = \sum_{\ell=0}^{n} \mathit{sign}(a_\ell)$. The *height* of w, denoted by $\mathit{height}(w)$, is the maximal counter value of any of its prefixes, *i.e.*, $\mathit{height}(w) = \max_{u \in \mathit{Pref}(w)} cv(u)$.

Definition 5.2.4 (Counter value and height)**.** Let $w \in \Sigma^*$. The *counter value of w according to $\mathcal{A}$*, denoted by $cv^{\mathcal{A}}(w)$, is the counter value n such that

$(q_0, 0) \xrightarrow{w} (q, n) \in \mathit{cruns}(\mathcal{A})$.
The *height of w according to $\mathcal{A}$*, denoted by $\mathit{height}^{\mathcal{A}}(w)$, is the maximal counter value among the prefixes of w, *i.e.*,

$$\mathit{height}^{\mathcal{A}}(w) = \max_{u \in \mathit{Pref}(w)} \mathit{cv}^{\mathcal{A}}(u).$$

A word w is accepted if there is a counted run from the *initial configuration* $(q_0, 0)$ to some configuration $(q, 0)$ such that $q \in F$. We highlight that $\mathit{cv}^{\mathcal{A}}(w) = 0$. The definition of language of $\mathcal{A}$ naturally follows.

Definition 5.2.5 (Realtime one-counter language)**.** The language accepted by a sound ROCA $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q \in F : (q_0, 0) \xrightarrow{w} (q, 0)\}.$$

A language L is called a *realtime one-counter language* (*ROCL*, for short) if there is a sound ROCA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

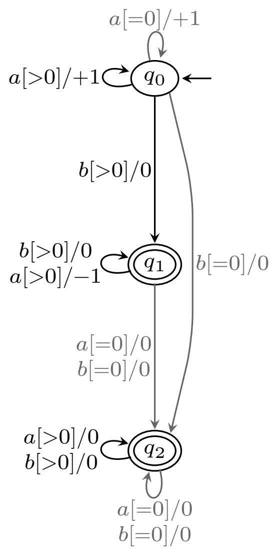
Example 5.2.6. Let $\mathcal{A}$ be the sound ROCA of Figure 5.1, which is repeated in the margin. A sample counted run is:

$$\begin{aligned}(q_0, 0) &\xrightarrow[+1]{a} (q_0, 1) \xrightarrow[+1]{a} (q_0, 2) \xrightarrow[0]{b} (q_1, 2) \xrightarrow[-1]{a} (q_1, 1) \\ &\xrightarrow[0]{b} (q_1, 1) \xrightarrow[-1]{a} (q_1, 0) \xrightarrow[0]{a} (q_2, 0).\end{aligned}$$

This counted run is accepting and thus $aababaa$ is accepted by $\mathcal{A}$. Moreover, $\mathit{cv}^{\mathcal{A}}(aababaa) = 0$ and $\mathit{height}^{\mathcal{A}}(aababaa) = 2$.
One can verify that the language of $\mathcal{A}$ is comprised of the words of the shape

$$a^{\ell} \cdot b \cdot b^{k_1} \cdot a \cdot b^{k_2} \cdot a \cdots b^{k_{\ell}} \cdot a \cdot u$$

with $\ell, k_1, k_2, \dots, k_{\ell} \geq 0$, and $u \in \{a, b\}^*$.



Assumption 5.2.7. In order to avoid having to treat particular cases, we assume from now on that any ROCL is non-empty.

We now introduce a refinement of the Myhill-Nerode congruence (see Definition 2.2.6) for a ROCL L accepted by some sound ROCA $\mathcal{A}$. Classes are refined according to the counter value of the words. That is, for two words u, v, we require that $\mathit{cv}^{\mathcal{A}}(u \cdot w) = \mathit{cv}^{\mathcal{A}}(v \cdot w)$ for any word w such that $u \cdot w$ and $v \cdot w$ are both in $\mathit{Pref}(L)$. The condition over the prefix of the language is to avoid refining the classes too much: if any extension of $u \cdot w$ (for instance) is never in L, the exact counter value is not useful for a learning algorithm. In other words, the words that are not in the prefix of L remain equivalent. We highlight that the definition depends on $\mathcal{A}$, due to the counter values of the words.

Definition 2.2.6. For a language $L \subseteq \Sigma^*$ and two words $u, v \in \Sigma^*$, we write $u \sim_L v$ if for all $w \in \Sigma^*$, we have $uw \in L \Leftrightarrow vw \in L$.

Definition 5.2.8 (Refinement of Myhill-Nerode)**.** Let $\mathcal{A}$ be an ROCA accepting the ROCL $L \subseteq \Sigma$. We define the congruence relation $\sim_{\mathcal{A}}$ over Σ^* such

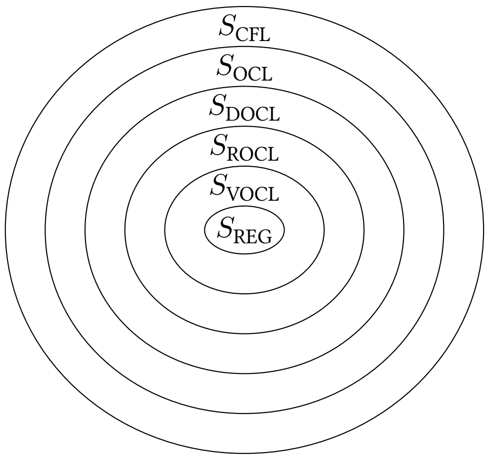


Figure 5.2: One-counter languages hierarchy extended from Figure 4.4 with ROCLs. Each S_i designates the set of all languages of type i.

that $u \sim_{\mathcal{A}} v$ if and only if:

$$\forall w \in \Sigma^* : uw \in L \Leftrightarrow vw \in L,$$
$$\forall w \in \Sigma^* : uw, vw \in \mathit{Pref}(L) \Rightarrow \mathit{cv}^{\mathcal{A}}(uw) = \mathit{cv}^{\mathcal{A}}(vw).$$

It is easy to check that $\sim_{\mathcal{A}}$ is a congruence.

Finally, we define the concept of *bounded language*. That is, given a language L and a counter limit ℓ, we restrict L to the words for which the height never exceeds ℓ. More precisely, as the height can only be defined via an ROCA accepting L, we define the bounded language of an ROCA. We explain in Section 5.4 how to construct a DFA accepting this bounded language.

Definition 5.2.9 (Bounded language)**.** Let $\mathcal{A}$ be an ROCA. The *bounded language* of $\mathcal{A}$ up to ℓ, denoted by $\mathcal{L}_{\leq \ell}(\mathcal{A})$, is

$$\mathcal{L}_{\leq \ell}(\mathcal{A}) = \{w \in \mathcal{L}(\mathcal{A}) \mid \mathit{height}^{\mathcal{A}}(w) \leq \ell\}.$$

5.2.2. Extended hierarchy of one-counter languages

Let us now extend the hierarchy of one-counter languages started in Section 4.3.2 by adding the set of ROCLs. We argue that ROCLs are strictly between VOCLs and DOCLs. In other words, allowing ε-transitions in a one-counter automaton allows one to encode more complex languages. Figure 5.2 gives the resulting hierarchy.

First, we state that ROCLs form a strict superset of VOCLs. The proof is actually the same as the proof of Proposition 4.3.9, which we do not repeat here.

Proposition 5.2.10. *Any VOCL is a ROCL but there exists a ROCL that is not a VOCL.*

Fischer *et al.* [FMR68] showed that requiring that the automaton takes n steps

[FMR68]: Fischer et al. (1968), "Counter Machines and Counter Languages"

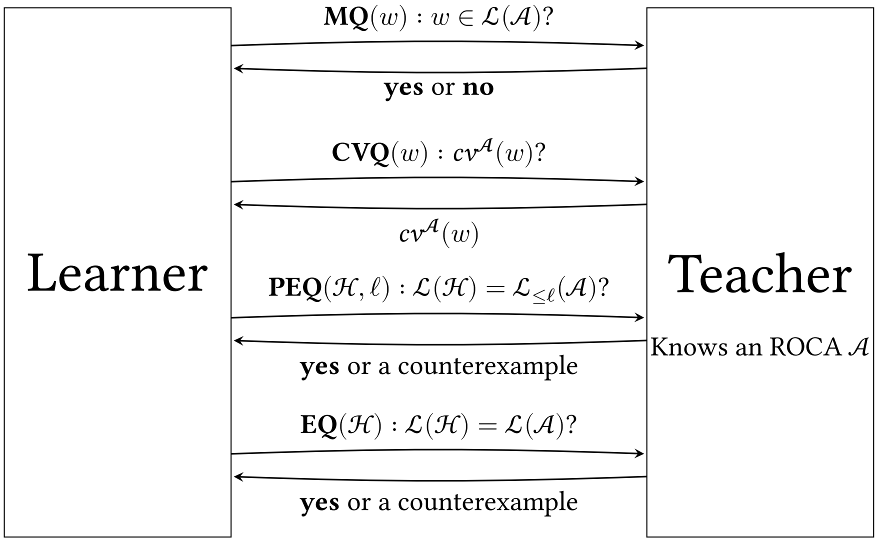
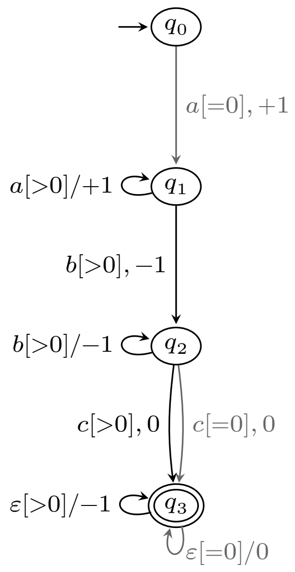

Learner
Teacher
Knows an ROCA $\mathcal{A}$
MQ$(w) : w \in \mathcal{L}(\mathcal{A})$?
yes or **no**
CVQ$(w) : cv^{\mathcal{A}}(w)$?
$cv^{\mathcal{A}}(w)$
PEQ$(\mathcal{H}, \ell) : \mathcal{L}(\mathcal{H}) = \mathcal{L}_{\leq \ell}(\mathcal{A})$?
yes or a counterexample
EQ$(\mathcal{H}) : \mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{A})$?
yes or a counterexample

Figure 5.3: Adaptation of Angluin's framework for ROCAs.

to process a word of length n reduces the expressivity compared to allowing any number of steps $\geq n$. However, they do not require that the counter is zero for a word to be accepted. We thus here give a slightly adjusted (sketch of) proof.



Proposition 5.2.11. *Any ROCL is a DOCL but there exists a DOCL that is not a ROCL.*

Sketch of proof. By definition, any ROCA is a DOCA. Hence, every ROCL is a DOCL.
Let $L = \{a^n b^m c \mid 0 < m \leq n\}$. Figure 4.1 (repeated in the margin) gives a DOCA accepting L, *i.e.*, L is a DOCL. One can show that there exists no ROCA accepting L using arguments similar to those presented in the proof of Proposition 4.3.10. In short, as the counter value of any accepted word must be zero, it follows that there must exist a loop $(p, c) \xrightarrow{b^\ell} (p, c)$ with $0 \leq c \leq 1$ and such that $(p, c) \xrightarrow{c} (q_f, 0) \in \mathit{cruns}(\mathcal{A})$ for some $q_f \in F^{\mathcal{A}}$. Hence, we can pump on that loop and show that $\mathcal{A}$ has to accept words that are not in L. □

5.2.3. Learning framework

For our learning algorithm for ROCAs, we adapt the framework of the learning algorithm for VOCLs (see Definition 4.3.22), which itself is an adaptation of Angluin's framework [Ang87]. As we do not consider pushdown alphabets, we cannot deduce the counter value of a word without running it through an ROCA. Hence, on top of membership and (partial) equivalence queries, we allow *counter value* queries to retrieve the counter value of some word according to the teacher's ROCA. This means that the teacher must know a sound ROCA that accepts L. Figure 5.3 gives a visual representation of the adapted Angluin's framework.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

Definition 5.2.12 (Queries for ROCAs)**.** Let $\mathcal{A}$ be the sound ROCA of the teacher. A learner for ROCAs can use four queries:

- A *membership query*, denoted by **MQ**(w), with $w \in \Sigma^*$, returns whether $w \in \mathcal{L}(\mathcal{A})$.
- A *counter value query*, denoted by **CVQ**(w), with $w \in \mathit{Pref}(\mathcal{L}(\mathcal{A}))$, returns $cv^{\mathcal{A}}(w)$.
- A *partial equivalence query*, denoted by **PEQ**$(\mathcal{H}, \ell)$, with $\mathcal{H}$ a DFA over Σ and $\ell \in \mathbb{N}$, returns
  - **yes** if $\mathcal{L}(\mathcal{H}) = \mathcal{L}_{\leq \ell}(\mathcal{A})$,
  - or a word w such that $w \in \mathcal{L}(\mathcal{H}) \Leftrightarrow w \notin \mathcal{L}_{\leq \ell}(\mathcal{A})$.
- An *equivalence query* **EQ**$(\mathcal{H})$, with $\mathcal{H}$ an ROCA over Σ, returns
  - **yes** if $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{A})$,
  - or a word w such that $w \in \mathcal{L}(\mathcal{H}) \Leftrightarrow w \notin \mathcal{L}(\mathcal{A})$.

Remark 5.2.13. Given that different ROCAs may give different counter values to the same input word, one may wonder why a counter value query over w does not return a range $[\ell, h]$ such that $cv^{\mathcal{A}}(w) \in [\ell, h]$ for any ROCA $\mathcal{A}$. While an upper bound may be useful, the lower bound will always be zero. Indeed, for a given word w, one can always construct an ROCA $\mathcal{A}$ such that $cv^{\mathcal{A}}(w) = 0$ (by unraveling loops and using states to count up to a certain bound, for instance). In this thesis, we instead assume that the teacher has an ROCA and can provide a precise counter value.

The general ideas of our learning algorithm are similar to those of L^*_{VOCA} (see Section 4.3.4). We will also learn a fragment of the target language, bounded by some counter value. Then, from the learned DFA, we will extract multiple ROCAs. We first adapt the concept of behavior graph, which is an infinite deterministic automaton encoding the counted runs of an ROCA, and show that it has a periodic structure, yielding a finite representation. Hence, learning a large enough DFA will be enough to observe that periodic structure.

5.3. Behavior graph of a realtime one-counter automaton

In this section, we rely on the notion of behavior graph introduced in Definition 4.3.14 for VOCAs and adapt it to ROCAs. While we could define the behavior graph of a *language* for VOCLs, we here need the behavior graph of an *automaton*, as the counter value and height of a word depend on this automaton. As for VOCAs, this behavior graph is a potentially infinite deterministic automaton. Nevertheless, we aim to show that the behavior graph of an ROCA also possesses an ultimately periodic structure and, so, can be represented in finite memory, using a trick similar to the finite representation of the behavior graph of a VOCL (see Section 4.3.3).

Definition 4.3.14. Let $L \subseteq \widetilde{\Sigma}^*$ be a VOCL, $\sim_L$ be its Myhill-Nerode congruence, and $\mathcal{A}$ be the deterministic (in)finite automaton defined from $\sim_L$. The *behavior graph* of L is the tuple $BG(L) = (\widetilde{\Sigma}, Q^{BG(L)}, q_0^{BG(L)}, F^{BG(L)}, \delta^{BG(L)})$ obtained from $\mathcal{A}$ by restricting the set of states to $Q^{BG(L)} = \{[\![w]\!]_{\sim_L} \mid w \in \mathit{Pref}(L)\}$.

While the behavior graph of a VOCL is defined from the Myhill-Nerode congruence of the language, we rely on $\sim_{\mathcal{A}}$ to define the behavior graph of an

ROCA $\mathcal{A}$. Since $\sim_{\mathcal{A}}$ is a congruence, a (potentially infinite) deterministic automaton can be defined from it, as is done for the Myhill-Nerode congruence in Definition 2.2.9. For clarity, we state the complete definition here.

Definition 5.3.1 (Behavior graph)**.** Let $\mathcal{A}$ be a sound ROCA accepting the language $L \subseteq \Sigma^*$ and $\sim_{\mathcal{A}}$ be the refinement of the Myhill-Nerode congruence for $\mathcal{A}$. The *behavior graph of* $\mathcal{A}$ is the deterministic (in)finite automaton $BG(\mathcal{A}) = (\Sigma, Q^{BG(\mathcal{A})}, q_0^{BG(\mathcal{A})}, F^{BG(\mathcal{A})}, \delta^{BG(\mathcal{A})})$ where:

- $Q^{BG(\mathcal{A})} = \{[\![w]\!]_{\sim_{\mathcal{A}}} \mid w \in \mathit{Pref}(L)\}$,
- $q_0^{BG(\mathcal{A})} = [\![\varepsilon]\!]_{\sim_{\mathcal{A}}}$,
- $F^{BG(\mathcal{A})} = \{[\![w]\!]_{\sim_{\mathcal{A}}} \mid w \in L\}$,
- $\delta^{BG(\mathcal{A})} : Q^{BG(\mathcal{A})} \times \Sigma \rightharpoonup Q^{BG(\mathcal{A})}$ is the partial transition function such that, for all $[\![w]\!]_{\sim_{\mathcal{A}}}, [\![wa]\!]_{\sim_{\mathcal{A}}} \in Q^{BG(\mathcal{A})}$ and $a \in \Sigma$,
$$[\![w]\!]_{\sim_{\mathcal{A}}} \xrightarrow{a} [\![w \cdot a]\!]_{\sim_{\mathcal{A}}} \in \mathit{runs}(BG(\mathcal{A})).$$

Note that $Q^{BG(\mathcal{A})}$ is not empty by Assumption 5.2.7. We highlight that $BG(\mathcal{A})$ only contains states that are reachable from the initial state and co-reachable from a final state, and its transition function is partial.

As we did for VOCAs, we group together all equivalence classes sharing the same counter value in what we call a *level*. The width of $BG(\mathcal{A})$ is then the maximal number of classes across all levels.

Definition 5.3.2 (Level and width)**.** The *level* $\ell \in \mathbb{N}$ of $BG(\mathcal{A})$, denoted by $\mathit{level}(BG(\mathcal{A}), \ell)$ is the set of states of $BG(\mathcal{A})$ with counter value ℓ, *i.e.*,
$$\mathit{level}(BG(\mathcal{A}), \ell) = \{[\![w]\!]_{\sim_{\mathcal{A}}} \in Q^{BG(\mathcal{A})} \mid \mathit{cv}^{\mathcal{A}}(w) = \ell\}.$$
The *width* of $BG(\mathcal{A})$, denoted by $\mathit{width}(BG(\mathcal{A}))$, is the maximal size of any level:
$$\mathit{width}(BG(\mathcal{A})) = \max_{\ell \in \mathbb{N}} |\mathit{level}(BG(\mathcal{A}), \ell).|$$

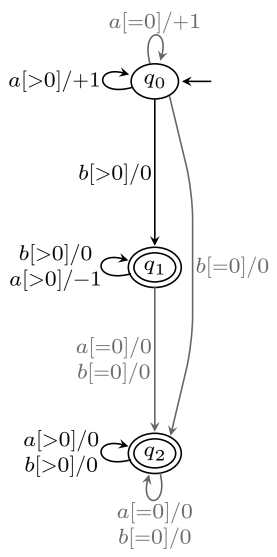
The next proposition states that $BG(\mathcal{A})$ accepts the same language as $\mathcal{A}$. It can be proved by a straightforward induction.

Proposition 5.3.3. *Let $\mathcal{A}$ be a sound ROCA and $BG(\mathcal{A})$ be its behavior graph. Then, $\mathcal{L}(BG(\mathcal{A})) = \mathcal{L}(\mathcal{A})$.*

Example 5.3.4. Let $\mathcal{A}$ be the ROCA of Figure 5.1, which is repeated in the margin. Example 5.2.6 gives the language L of $\mathcal{A}$. We can check that $b \sim_{\mathcal{A}} abba$:

- For all $w \in \Sigma^*$, we have $bw \in L$ if and only if $abbaw \in L$.
- For all $w \in \Sigma^*$ such that bw and $abbaw$ are both in $\mathit{Pref}(L)$, we have $\mathit{cv}^{\mathcal{A}}(bw) = \mathit{cv}^{\mathcal{A}}(abbaw)$.

However, $ab \not\sim_{\mathcal{A}} aab$ since ab and aab are both in $\mathit{Pref}(L)$ but $\mathit{cv}^{\mathcal{A}}(ab) = 1$ while $\mathit{cv}^{\mathcal{A}}(aab) = 2$.
The behavior graph $BG(\mathcal{A})$ of $\mathcal{A}$ is given in Figure 5.4. As in Example 4.3.18, this behavior graph is ultimately periodic and has thus a finite representa-



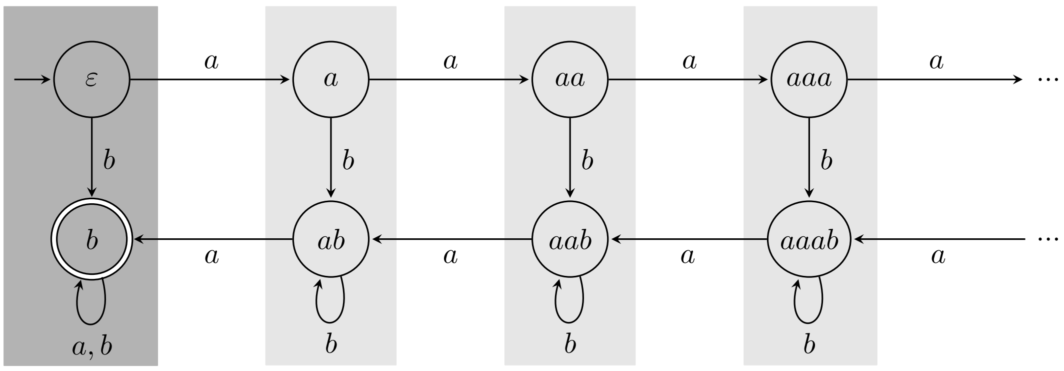


Figure 5.4: Behavior graph of the ROCA of Figure 5.1.

tion.

5.3.1. Finite representation

We now proceed on proving that there always exists a finite representation for the behavior graph of any ROCA. First, we show that the width of $BG(\mathcal{A})$ is always bounded by the number of states of $\mathcal{A}$.

Lemma 5.3.5. *Let $BG(\mathcal{A})$ be the behavior graph of a sound ROCA $\mathcal{A}$. Then, $\mathit{width}(BG(\mathcal{A})) \leq |Q^{\mathcal{A}}|$.*

Proof. Let $L = \mathcal{L}(\mathcal{A})$. Let w and w' be two words such that

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{w} (q, n) \in \mathit{cruns}(\mathcal{A})$$

and

$$(q_0^{\mathcal{A}}, 0) \xrightarrow{w'} (q, n) \in \mathit{cruns}(\mathcal{A}).$$

Observe that both counted runs end in the same configuration. Hence, $\mathit{cv}^{\mathcal{A}}(w) = \mathit{cv}^{\mathcal{A}}(w')$ and $w \in \mathit{Pref}(L)$ if and only if $w' \in \mathit{Pref}(L)$. Moreover, for any $v \in \Sigma^*$, we have $wv \in L$ if and only if $w'v \in L$. There are two cases:

- If $w, w' \in \mathit{Pref}(L)$, then we have $w \sim_{\mathcal{A}} w'$ as $\mathit{cv}^{\mathcal{A}}(w) = \mathit{cv}^{\mathcal{A}}(w')$.
- If $w, w' \notin \mathit{Pref}(L)$, then we have $w \sim_{\mathcal{A}} w'$, by definition.

In every case, $w \sim_{\mathcal{A}} w'$. So, there cannot be more than $|Q^{\mathcal{A}}|$ states per level. That is, $\mathit{width}(BG(\mathcal{A})) \leq |Q^{\mathcal{A}}|$. □

This allows us to describe the behavior graph by enumerating the states of each level and defining the transitions using these enumerations, as was done for VOCAs in Section 4.3.3. More precisely, for each level ℓ of $BG(\mathcal{A})$, we define an enumeration

$$\nu_\ell : \mathit{level}(BG(\mathcal{A}), \ell) \to \{1, \dots, K\},$$

with $K = \mathit{width}(BG(\mathcal{A}))$.

Once the enumerations are fixed, we have to encode the transitions. This requires more care than for VOCAs, as we now have to explicitly take into account the counter operation. That is, the counter operation of the transition must also be stored, as it cannot be deduced from the symbol anymore. For each level ℓ, the transitions are encoded via a mapping

$$\tau_\ell : \{1, \dots, K\} \times \Sigma \rightharpoonup \{1, \dots, K\} \times \{-1, 0, +1\}$$

such that for all $i \in \{1, \dots, K\}$ and $a \in \Sigma$, $\tau_\ell(i, a) = (j, c)$ if there exist $[\![u]\!]_{\sim_\mathcal{A}}, [\![ua]\!]_{\sim_\mathcal{A}} \in Q^{BG(\mathcal{A})}$ such that

- ▶ $cv^\mathcal{A}(u) = \ell$,
- ▶ $cv^\mathcal{A}(ua) = \ell + c$,
- ▶ $\nu_\ell([\![u]\!]_{\sim_\mathcal{A}}) = i$, and
- ▶ $\nu_{\ell+c}([\![ua]\!]_{\sim_\mathcal{A}}) = j$,

and $\tau_\ell(i, a)$ is left undefined otherwise.

We thus obtain a *description* α of $BG(\mathcal{A})$ by listing each transition mapping:

$$\alpha = \tau_0 \tau_1 \tau_2 \dots$$

It remains to show that there always exists an ultimately periodic description, as stated in the next theorem, which is the ROCA counterpart of Theorem 4.3.19. The proof relies on an isomorphism we establish (in the next section) between the behavior graph of an ROCA $\mathcal{A}$ and that of a suitable VOCA constructed from $\mathcal{A}$, which is detailed in the next section. Once this isomorphism is obtained, the theorem is immediate, thanks to Theorem 4.3.19. Finally, Section 5.3.3 explains how to construct an ROCA from an ultimately periodic description of a behavior graph.

Theorem 4.3.19. Let $BG(L)$ be the behavior graph of a VOCL L. Then, there exists, for each level ℓ, an enumeration $\nu_\ell : level(BG(L), \ell) \to \{1, \dots, width(BG(L))\}$, such that the corresponding description α of $BG(L)$ is an ultimately periodic word with offset $m > 0$ and period $k \geq 0$, i.e., $\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega$.

Theorem 5.3.6. *Let $BG(\mathcal{A})$ be the behavior graph of a sound ROCA $\mathcal{A}$. Then, there exist, for each level ℓ, an enumeration*

$$\nu_\ell : level(BG(\mathcal{A}), \ell) \to \{1, \dots, width(BG(\mathcal{A}))\},$$

such that the corresponding description α of $BG(\mathcal{A})$ is an ultimately periodic word with offset $m > 0$ and period $k \geq 0$, i.e.,

$$\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega.$$

5.3.2. Isomorphism with the behavior graph of a visibly one-counter language

In order to show that the behavior graph of a sound ROCA $\mathcal{A}$ accepting L is isomorphic to the behavior graph of some VOCL (see Definition 4.3.14), we first explain how to construct a VOCA accepting a VOCL $\widetilde{L}$ encoding the counter operations of the transitions of $\mathcal{A}$. For instance, every transition $q \xrightarrow[+1]{a} \in runs(\mathcal{A})$ becomes a transition reading a_c in the VOCA.[2]

2: Observe that we add a subscript c to a.

Let us start by defining the pushdown alphabet $\widetilde{\Sigma}$ that will be used in the constructed VOCA. In short, every symbol of Σ gives birth to a call symbol, a return symbol, and an internal symbol. That is, let $\widetilde{\Sigma} = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$ with

$$\Sigma_c = \{a_c \mid a \in \Sigma\},$$
$$\Sigma_r = \{a_r \mid a \in \Sigma\},$$

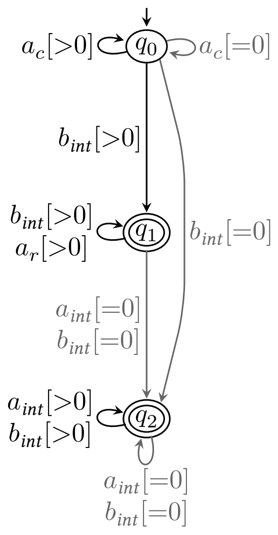
and

$$\Sigma_{int} = \{a_{int} \mid a \in \Sigma\}.$$

We then define a VOCA $\widetilde{\mathcal{A}}$ over $\widetilde{\Sigma}$. The set of states, initial state, and the set of final states are directly copied from the ROCA $\mathcal{A}$. The transitions require more care as we have to use the symbols from $\widetilde{\Sigma}$. For all $q \xrightarrow[o]{a[g]} p \in \mathit{runs}(\mathcal{A})$, we define in $\widetilde{\mathcal{A}}$ a transition $q \xrightarrow{b[g]} p$ such that

$$b = \begin{cases} a_c & \text{if } o = +1 \\ a_r & \text{if } o = -1 \\ a_{int} & \text{if } o = 0. \end{cases}$$

Example 5.3.7. We consider again the ROCA $\mathcal{A}$ given in Figure 5.1. The VOCA $\widetilde{\mathcal{A}}$ constructed from $\mathcal{A}$ is given in Figure 4.3 and is repeated in the margin, for convenience.



Let us now move towards describing the language of $\widetilde{\mathcal{A}}$. Namely, we argue that for every word w over Σ, we can construct a word $\widetilde{w}$ over $\widetilde{\Sigma}$ such that $\mathcal{A}$ and $\widetilde{\mathcal{A}}$ agree on the counter value of w and $\widetilde{w}$, and on whether the word is accepted. To do so, we introduce a mapping $\lambda_{\mathcal{A}} : \Sigma^* \rightharpoonup \widetilde{\Sigma}^*$. Given a word $u = a_1 \cdots a_k$, its corresponding word $\lambda_{\mathcal{A}}(u) = \widetilde{a_0} \cdots \widetilde{a_k}$ is constructed such that for each $i \in \{1, \dots, k\}$:

$$\widetilde{a_i} = \begin{cases} b_c & \text{if } \mathit{cv}^{\mathcal{A}}(a_1 \cdots a_i) > \mathit{cv}^{\mathcal{A}}(a_1 \cdots a_{i-1}) \text{ and } a_i = b \\ b_r & \text{if } \mathit{cv}^{\mathcal{A}}(a_1 \cdots a_i) < \mathit{cv}^{\mathcal{A}}(a_1 \cdots a_{i-1}) \text{ and } a_i = b \\ b_{int} & \text{if } \mathit{cv}^{\mathcal{A}}(a_1 \cdots a_i) = \mathit{cv}^{\mathcal{A}}(a_1 \cdots a_{i-1}) \text{ and } a_i = b. \end{cases}$$

Note that given $\lambda_{\mathcal{A}}(u) \in \widetilde{\Sigma}^*$, it is easy to get u back: simply discard the index in $\{c, r, \mathit{int}\}$ of each symbol of $\lambda_{\mathcal{A}}(u)$. We denote by $\widetilde{\lambda}$ this mapping from $\widetilde{\Sigma}$ to Σ, *i.e.*, $\widetilde{\lambda}(a_x) = a$ for $a \in \Sigma$ and $x \in \{c, r, \mathit{int}\}$.[3]

3: Notice that $\widetilde{\lambda}$ does not depend on $\mathcal{A}$ nor on $\widetilde{\mathcal{A}}$.

Example 5.3.8. Let us continue Example 5.3.7. We construct $\lambda_{\mathcal{A}}(u)$ for $u = a \cdot a \cdot b \cdot a \cdot b \cdot a \cdot a$. Recall from Example 5.2.6 that we have the following counted run for u in $\mathcal{A}$:

$$\begin{aligned}(q_0, 0) &\xrightarrow{a} (q_0, 1) \xrightarrow{a} (q_0, 2) \xrightarrow{b} (q_1, 2) \xrightarrow{a} (q_1, 1) \\ &\xrightarrow{b} (q_1, 1) \xrightarrow{a} (q_1, 0) \xrightarrow{a} (q_2, 0).\end{aligned}$$

By encoding the counter operations into the symbols, we thus obtain the word $\lambda_{\mathcal{A}}(u) = a_c \cdot a_c \cdot b_{int} \cdot a_r \cdot b_{int} \cdot a_r \cdot a_{int}$, which yields the following counted run in $\widetilde{\mathcal{A}}$:

$$(q_0, 0) \xrightarrow{a_c} (q_0, 1) \xrightarrow{a_c} (q_0, 2) \xrightarrow{b_{int}} (q_1, 2) \xrightarrow{a_r} (q_1, 1) \\ \xrightarrow{b_{int}} (q_1, 1) \xrightarrow{a_r} (q_1, 0) \xrightarrow{a_{int}} (q_2, 0).$$

The function $\lambda_{\mathcal{A}}$ allows us to easily define the language of $\widetilde{\mathcal{A}}$ as the set of all words $\lambda_{\mathcal{A}}(u)$ with $u \in \mathcal{L}(\mathcal{A})$. Moreover, $\widetilde{\mathcal{A}}$ preserves the set of prefixes of $\mathcal{L}(\mathcal{A})$ and the counter value of the words. These properties are easily proved by construction.

Lemma 5.3.9. *Let $\mathcal{A}$ be a sound ROCA and $\widetilde{\mathcal{A}}$ be its corresponding VOCA. We have:*

- $\forall u \in \Sigma^* : \mathit{cv}^{\mathcal{A}}(u) = \mathit{cv}(\lambda_{\mathcal{A}}(u))$,
- $\mathcal{L}(\widetilde{\mathcal{A}}) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathcal{L}(\mathcal{A})\}$, *and*
- $\mathit{Pref}(\mathcal{L}(\widetilde{\mathcal{A}})) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathit{Pref}(\mathcal{L}(\mathcal{A}))\}$.

We can now clearly state the isomorphism between $BG(\mathcal{A})$ and $BG(\mathcal{L}(\widetilde{\mathcal{A}}))$. First of all, we need to translate the symbols between the two graphs, *i. e.*, the isomorphism is up to $\lambda_{\mathcal{A}}$ and $\widetilde{\lambda}$ depending on the direction. We have to show that $\sim_{\mathcal{A}}$ and $\sim_{\mathcal{L}(\widetilde{\mathcal{A}})}$ agree (*i. e.*, they define the same equivalence classes up to $\lambda_{\mathcal{A}}$ and $\widetilde{\lambda}$), which can be obtained by exploiting the definitions and properties of the two relations (see Definitions 2.2.6 and 5.2.8, and Lemma 4.3.13). Furthermore, by the previous lemma, we know that $\lambda_{\mathcal{A}}$ respects the counter values. Hence, the class of w is part of the level ℓ of $BG(\mathcal{A})$ if and only if the class of $\lambda_{\mathcal{A}}(w)$ is also on the level ℓ of $BG(\mathcal{L}(\widetilde{\mathcal{A}}))$. The proof is given in Section A.1.

Theorem 5.3.10. *Let $\mathcal{A}$ be a sound ROCA, $BG(\mathcal{A})$ be its behavior graph, $\widetilde{\mathcal{A}}$ be the corresponding VOCA accepting $\widetilde{L}$, and $BG(\widetilde{L})$ be the behavior graph of $\widetilde{L}$. Then,*

- *$BG(\mathcal{A})$ and $BG(\widetilde{L})$ are isomorphic up to $\lambda_{\mathcal{A}}$ and $\widetilde{\lambda}$, and*
- *the isomorphism respects the counter values (i. e., level membership) and both offset and period of periodic descriptions.*

Hence, if α is a periodic description of $BG(\mathcal{A})$ with offset m and period k, then by $\lambda_{\mathcal{A}}$ we get a periodic description of $BG(\widetilde{L})$ with the same offset and period. The converse is also true by using $\widetilde{\lambda}$. That is, we immediately obtain that there exists a finite representation (via an ultimately periodic description) of $BG(\mathcal{A})$, *i. e.*, Theorem 5.3.6 follows from the previous theorem and Theorem 4.3.19.

5.3.3. Constructing a realtime one-counter automaton from a description

Finally, let us explain how to construct an ROCA $\mathcal{A}_\alpha$ from an ultimately periodic description α of a behavior graph $BG(\mathcal{A})$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_\alpha)$. While this is not interesting in itself (as $\mathcal{A}$ is already known here), it will be useful during the learning algorithm. Indeed, the same algorithm will be applied to descriptions derived from a fragment of $BG(\mathcal{A})$.

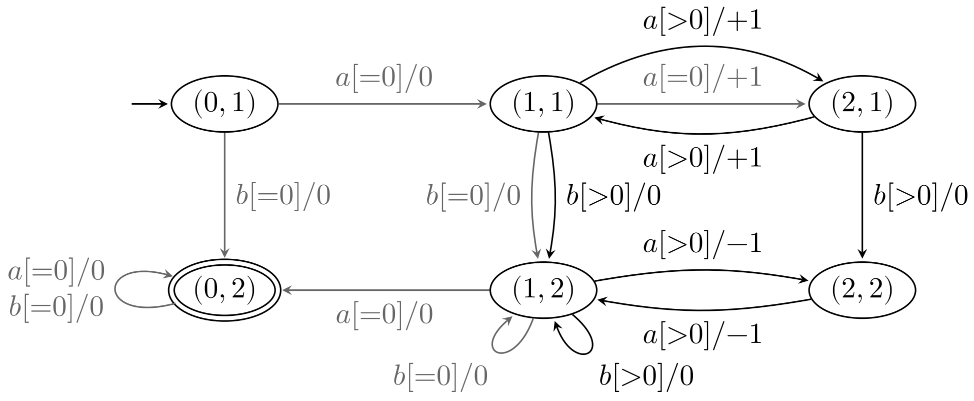
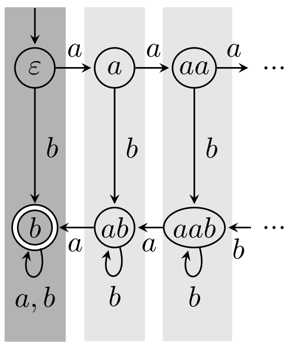


Figure 5.5: The ROCA constructed from a periodic description of the behavior graph of Figure 5.4.

Let us provide an example giving the intuition of the construction.



Example 5.3.11. Let $\mathcal{A}$ be the ROCA from Figure 5.1 and $BG(\mathcal{A})$ be its behavior graph from Figure 5.4 (which is repeated in the margin). Recall that the width of $BG(\mathcal{A})$ is 2. Let us assume that

$$\nu_0([\![\varepsilon]\!]_{\sim_{\mathcal{A}}}) = \nu_1([\![a]\!]_{\sim_{\mathcal{A}}}) = \cdots = 1$$
$$\nu_0([\![b]\!]_{\sim_{\mathcal{A}}}) = \nu_1([\![ab]\!]_{\sim_{\mathcal{A}}}) = \ldots = 2.$$

We then have the following τ_ℓ mappings:

$$\begin{array}{ll}
\tau_0(1,a) = (1,+1) & \tau_0(1,b) = (2,0) \\
\tau_0(2,a) = (2,0) & \tau_0(2,b) = (2,0) \\
\tau_1(1,a) = (1,+1) & \tau_1(1,b) = (2,0) \\
\tau_1(2,a) = (2,-1) & \tau_1(2,b) = (2,0) \\
\tau_2(1,a) = (1,+1) & \tau_2(1,b) = (2,0) \\
\tau_2(2,a) = (2,-1) & \tau_2(2,b) = (2,0) \\
\vdots & \vdots
\end{array}$$

Let $\alpha = \tau_0(\tau_1\tau_2)^\omega$ be a periodic description of $BG(\mathcal{A})$.
We construct an ROCA $\mathcal{A}_\alpha$. The idea is to encode in the states the current level ℓ and number $\nu_\ell(q)$ associated to the state $q \in Q^{BG(\mathcal{A})}$ in the states of $\mathcal{A}_\alpha$. Moreover, the counter value is never modified while in the levels forming the initial part of α (here, only the level 0, as $m = 1$). However, the counter value is modified every time we go from a level to another one in the periodic part. That is,

$$Q^{\mathcal{A}_\alpha} = \{0,1,2\} \times \{1,2\}.$$

Then, the transitions from a state $(0,\cdot)$ (*i.e.*, at level 0 in the behavior graph) are easily defined by simply copying the information stored in τ_0. The transitions from states $(1,\cdot), (2,\cdot)$ (*i.e.*, at levels 1 and 2) need to modify the counter. The resulting ROCA $\mathcal{A}_\alpha$ is given in Figure 5.5. We can clearly see that $\mathcal{A}_\alpha$ mimics the periodic description as the states $(1,0)$ and $(2,0)$ form the initial part that is followed by a repeating part formed by the states

$(1,1),(2,1),(1,2)$ and $(2,2)$.

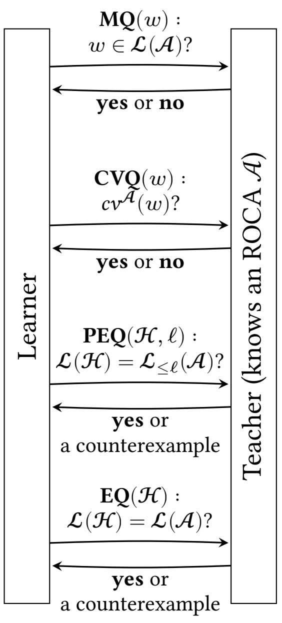
The following proposition states the existence of such an ROCA $\mathcal{A}_\alpha$ and its size. The formal proof is deferred to Section A.2.

Proposition 5.3.12. *Let $BG(\mathcal{A})$ be the behavior graph of some sound ROCA $\mathcal{A}$ and $\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega$ be an ultimately periodic description of $BG(\mathcal{A})$ with offset m and period k. Then, one can construct an ROCA $\mathcal{A}_\alpha$ from α such that*

- *$\mathcal{L}(\mathcal{A}_\alpha) = \mathcal{L}(\mathcal{A})$, and*
- *the size of $\mathcal{A}_\alpha$ is polynomial in m, k and width$(BG(\mathcal{A}))$.*

5.4. Learning algorithm

In this section, we introduce L^*_{ROCA}, a learning algorithm for ROCAs, using the framework defined in Section 5.2.3. That is, we assume that the teacher knows a sound ROCA $\mathcal{A}$ accepting a language L, and we have four types of queries: membership queries (**MQ**), counter value queries (**CVQ**), partial equivalence queries (**PEQ**), and equivalence queries (**EQ**). See Definition 5.2.12 for their definitions. We repeat the visual representation of the adapted Angluin's framework in the margin.



MQ(w) :
$w \in \mathcal{L}(\mathcal{A})$?
yes or **no**
CVQ(w) :
$cv^{\mathcal{A}}(w)$?
yes or **no**
PEQ$(\mathcal{H}, \ell)$:
$\mathcal{L}(\mathcal{H}) = \mathcal{L}_{\leq \ell}(\mathcal{A})$?
yes or
a counterexample
EQ$(\mathcal{H})$:
$\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{A})$?
yes or
a counterexample
Learner
Teacher (knows an ROCA $\mathcal{A}$)

In order to learn the language L by using these queries, the learner will learn a periodic description α of the behavior graph $BG(\mathcal{A})$. From α, an ROCA accepting L can be constructed, thanks to Proposition 5.3.12. More precisely, the idea is similar to L^*_{VOCA} (see Section 4.3.4): we learn $BG(\mathcal{A})$ up to a counter limit (*i.e.*, we learn a DFA) from which we extract every possible ultimately periodic description, each yielding an ROCA. If one of these ROCAs is correct, we are done. Otherwise, we need to increase the counter limit and repeat the procedure.

We start by properly defining this bounded behavior graph (as was done for VOCAs), before introducing the adaptation of an observation table in Section 5.4.2. Then, Section 5.4.3 explains how to obtain a right-congruence from a table, allowing us to construct DFA and ROCA hypotheses in Section 5.4.4. We give the main loop of our algorithm and a way to process counterexamples in Section 5.4.5. Finally, we give a complete example of an execution of the learning algorithm in Section 5.4.6. First, we give the following theorem that summarizes our results. Its proof can be found in Section A.8.

Theorem 5.4.1. *Let $\mathcal{A}$ be the sound ROCA of the teacher and ζ be the length of the longest counterexample returned by the teacher on (partial) equivalence queries. Then,*

- *the L^*_{ROCA} algorithm eventually terminates and returns an ROCA accepting $\mathcal{L}(\mathcal{A})$ and whose size is polynomial in $|Q^{\mathcal{A}}|$ and $|\Sigma|$,*
- *in time and space exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ, and*
- *asking a number of* **PEQ** *in $\mathcal{O}(\zeta^3)$, a number of* **EQ** *in $\mathcal{O}(|Q^{\mathcal{A}}|\zeta^2)$, and a number of* **MQ** *and* **CVQ** *exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ.*

5.4.1. Bounded behavior graph

We first define what the learner actually learns: the *bounded* behavior graph up to some counter limit ℓ, which is a DFA accepting a subset of $\mathcal{L}(\mathcal{A})$ such that the heights of the words that can be read in the DFA never exceeds ℓ.

Definition 5.4.2 (Bounded behavior graph)**.** Let $BG(\mathcal{A})$ be a behavior graph of some sound ROCA $\mathcal{A}$. The *bounded behavior graph up to* ℓ is the DFA $BG_{\leq \ell}(\mathcal{A})$ obtained from $BG(\mathcal{A})$ such that

$$Q^{BG_{\leq \ell}(\mathcal{A})} = \{[\![w]\!]_{\sim_\mathcal{A}} \in Q^{BG(\mathcal{A})} \mid \mathit{height}^\mathcal{A}(w) \leq \ell\}.$$

The values of the (partial) function $\delta^{BG_{\leq \ell}(\mathcal{A})}$, the initial state, and the final states are naturally defined over the subgraph.
The language of $BG_{\leq \ell}(\mathcal{A})$, denoted by $L_{\leq \ell}$, is

$$\begin{aligned} L_{\leq \ell} &= \mathcal{L}_{\leq \ell}(\mathcal{A}) \\ &= \{w \in \mathcal{L}(\mathcal{A}) \mid \mathit{height}^\mathcal{A}(w) \leq \ell\} \\ &= \{w \in \mathcal{L}(\mathcal{A}) \mid \forall v \in \mathit{Pref}(w) : 0 \leq \mathit{cv}^\mathcal{A}(v) \leq \ell\}. \end{aligned}$$

Notice that, by definition, each state of $BG_{\leq \ell}(\mathcal{A})$ is reachable. The next lemma gives an upper bound on the number of states.

Lemma 5.4.3. *The number of states of $BG_{\leq \ell}(\mathcal{A})$ is at most $(\ell + 1) \cdot |Q^\mathcal{A}|$.*

Proof. By Lemma 5.3.5, each level of $BG(\mathcal{A})$ has at most $|Q^\mathcal{A}|$ states. Therefore, $BG_{\leq \ell}(\mathcal{A})$ has at most $(\ell + 1) \cdot |Q^\mathcal{A}|$ states. □

Lemma 5.3.5. The width of $BG(\mathcal{A})$ is at most $|Q^\mathcal{A}|$.

We can thus immediately bound the number of equivalence classes $[\![w]\!]_{\sim_\mathcal{A}}$ with $\mathit{height}^\mathcal{A}(w) \leq \ell$. Indeed, all the words in $\Sigma^* \setminus \mathit{Pref}(L)$ are in a single class which is not present in $BG(\mathcal{A})$ (and, thus, not in $BG_{\leq \ell}(\mathcal{A})$ either).

Corollary 5.4.4. *The number of equivalence classes $[\![w]\!]_{\sim_\mathcal{A}}$ with $\mathit{height}^\mathcal{A}(w) \leq \ell$ is at most $(\ell + 1) \cdot |Q^\mathcal{A}| + 1$.*

Observe that ultimately periodic descriptions can be extracted from $BG_{\leq \ell}(\mathcal{A})$, similarly to what was done for VOCAs in Section 4.3.4. Moreover, if ℓ is big enough, then one of these descriptions match with an ultimately periodic description of $BG(\mathcal{A})$. Hence, it is sufficient to learn $BG_{\leq \ell}(\mathcal{A})$, starting with small values of ℓ and increasing it once we are sure that none of its descriptions are good. The rest of this section explains how to learn $BG_{\leq \ell}(\mathcal{A})$, and how and when to increase ℓ.

5.4.2. Observation table

As for L^*_{VOCA}, L^*_{ROCA}relies on an adaptation of the observation table of L^* to store the knowledge gathered during the learning process. More precisely, this table approximates the equivalence classes of $\sim_\mathcal{A}$ and therefore stores information about both membership to $\mathcal{L}(\mathcal{A})$ and counter values (for words known to be in $\mathit{Pref}(L)$). As we want to actually learn $BG_{\leq \ell}(\mathcal{A})$ from which ROCAs will be extracted, the table depends on a counter limit $\ell \in \mathbb{N}$. Furthermore, our table uses two sets of separators: $\widehat{S}$ which are used to store membership information, and S for counter value information. We define this formally, before giving an example.

Definition 5.4.5 (Observation table up to ℓ)**.** Let $\ell \in \mathbb{N}$ be a counter limit, $BG_{\leq \ell}(\mathcal{A})$ be the bounded behavior graph of a sound ROCA $\mathcal{A}$ up to ℓ, and $L_{\leq \ell}$ be the language of $BG_{\leq \ell}(\mathcal{A})$. An *observation table up to* ℓ is a tuple $\mathcal{O}_{\leq \ell} = (R, S, \widehat{S}, T, C)$ with:

- $R \subsetneq \Sigma^*$ a finite prefix-closed set of *representatives*,
- $S \subseteq \widehat{S} \subsetneq \Sigma^*$ two finite suffix-closed sets of *separators*,
- $T : (R \cup R\Sigma) \cdot \widehat{S} \to \{\mathbf{no}, \mathbf{yes}\}$, and
- $C : (R \cup R\Sigma) \cdot S \to \{0, \ldots, \ell\} \cup \{\bot\}$.

Let $\mathit{Pref}(\mathcal{O}_{\leq \ell})$ be the set of words that are known to be in the prefix of $L_{\leq \ell}$, *i. e.*,

$$\mathit{Pref}(\mathcal{O}_{\leq \ell}) = \{w \in \mathit{Pref}(u \cdot s) \mid u \in R \cup R\Sigma, s \in \widehat{S}, T(u \cdot s) = \mathbf{yes}\}.$$

For all $u \in R \cup R\Sigma$, we require that:

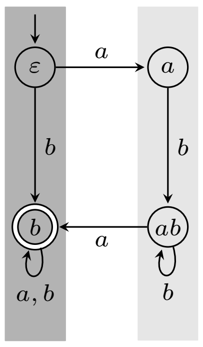
$$\forall s \in \widehat{S} : T(u \cdot s) = \begin{cases} \mathbf{yes} & \text{if } u \cdot s \in L_{\leq \ell} \\ \mathbf{no} & \text{otherwise,} \end{cases}$$

$$\forall s \in S : C(u \cdot s) = \begin{cases} cv^{\mathcal{A}}(u \cdot s) & \text{if } u \cdot s \in \mathit{Pref}(\mathcal{O}_{\leq \ell}) \\ \bot & \text{otherwise.} \end{cases}$$

Notice that the domains of T and C are different. We again highlight the fact that our data structure is different from the table used for VOCAs [NL10] and introduced in Section 4.3.4. The difference lies in our use of two sets of separators, due to the fact that we can not compute the counter value of a word directly from its symbols, but we need to ask a query to the teacher.[4] We argue below (in Example 5.4.8) why we need these two sets, once we know how to fill the table. In the next section, we explain how to use T and C to approximate $\sim_{\mathcal{A}}$ (up to the counter limit ℓ).

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

4: Recall that $\mathbf{CVQ}(w)$ can only be asked when w is in the prefix of $\mathcal{L}(\mathcal{A})$.



Example 5.4.6. Let $\mathcal{A}$ be the ROCA of Figure 5.1 and $\ell = 1$ be the counter limit we consider. Hence, we learn $BG_{\leq \ell}(\mathcal{A})$ whose set of states is given by the first two levels of $BG(\mathcal{A})$ from Figure 5.4 (see the figure in the margin). Figure 5.6 gives an observation table $\mathcal{O}_{\leq 1}$ up to $\ell = 1$. Its first column contains the elements of $R \cup R\Sigma$ such that the upper part is constituted by $R = \{\varepsilon, a, ab, aba, aa\}$ and the lower part by $R\Sigma \setminus R$. The first row contains the elements of $\widehat{S}$ such that the left part is constituted by $S = \{\varepsilon\}$ and the right part by $\widehat{S} \setminus S$.
For each element us in $(R \cup R\Sigma) \cdot S$, we store the two values $T(us)$ and $C(us)$ in the cell at the intersection of row u and column s. For instance, these values are 0 and $\bot$ for the row aa and column ε.
For each element us in $(R \cup R\Sigma) \cdot S$, we only need to store the value $T(us)$. Notice that $\mathit{Pref}(\mathcal{O}_{\leq \ell})$ is a proper subset of $\mathit{Pref}(L_{\leq \ell})$. For instance,

	ε	a	ba
ε	**no**, 0	**no**	**yes**
a	**no**, 1	**no**	**yes**
ab	**no**, 1	**yes**	**yes**
aba	**yes**, 0	**yes**	**yes**
aa	**no**, $\bot$	**no**	**no**
b	**yes**, 0	**yes**	**yes**
abb	**no**, 1	**yes**	**yes**
$abaa$	**yes**, 0	**yes**	**yes**
$abab$	**yes**, 0	**yes**	**yes**
aaa	**no**, $\bot$	**no**	**no**
aab	**no**, $\bot$	**no**	**no**

Figure 5.6: Example of an observation table.

$aababaa \notin Pref(\mathcal{O}_{\leq \ell})$ and $aababaa \in Pref(L_{\leq \ell})$.

Filling the table

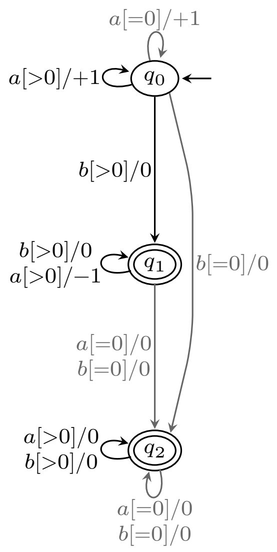
Let us explain how to fill the table $\mathcal{O}_{\leq \ell}$, using queries. Obtaining the value for $T(u \cdot s)$ (with $u \in R \cup R\Sigma$ and $s \in \widehat{S}$) is as follows: ask **MQ**$(u \cdot s)$ to learn whether $u \cdot s \in L$. Clearly, if $u \cdot s \notin L$, we can immediately put a **no** in the table, as $u \cdot s \notin L_{\leq \ell}$. However, we have to be more careful when $u \cdot s \in L$, as it may be that the height of $u \cdot s$ in $\mathcal{A}$ exceeds the limit ℓ. When $u \cdot s \in L$, every prefix x of $u \cdot s$ is in the prefix of L and we can thus ask **CVQ**(x). This allows us to determine whether $u \cdot s \in L_{\leq \ell}$, using one **MQ** and $|u \cdot s|$ **CVQ**, which in turn allows us to update the set $Pref(\mathcal{O}_{\leq \ell})$.

Assume that we learned that $u \cdot s \in Pref(\mathcal{O}_{\leq \ell})$ for some $u \in R \cup R\Sigma$ and $s \in S$ and we want to fill $C(u \cdot s)$. Since $u \cdot s \in Pref(\mathcal{O}_{\leq \ell})$, we know that $u \cdot s \in Pref(L)$ and we can ask **CVQ**$(u \cdot s)$. That is, determining $C(u \cdot s)$ requires at most a single **CVQ**, due to how $Pref(\mathcal{O}_{\leq \ell})$ is computed.

The next lemma is easily obtained given the above procedures, as $R \cup R\Sigma$ is prefix-closed and $\widehat{S}$ is suffix-closed.

Lemma 5.4.7. *Filling T and C requires a number of* **MQ** *and* **CVQ** *that is polynomial in the sizes of $R \cup R\Sigma$ and $\widehat{S}$.*

We now argue why it is necessary to use two sets of separators in Definition 5.4.5, via an example.



Example 5.4.8. Assume that we only use the set S in Definition 5.4.5 and that the current observation table is the leftmost table $\mathcal{O}_{\leq 1}$, given in Figure 5.7 for the ROCA from Figure 5.1 (which is repeated in the margin, for convenience). Moreover, assume that we use the classical L^* algorithm (see Section 3.2). As we can see, $\mathcal{O}_{\leq 1}$ is not closed since the row for $abb \in R\Sigma$ (*i.e.*, the row with contents **no**, $\bot$) does not appear in the upper part of the table. In other words, there is no $u \in R$ with the same row contents. So, we add abb in the upper part (*i.e.*, abb is now in R), and $abba$ and $abbb$ in the lower part, to

	ε
ε	**no**, 0
a	**no**, 1
ab	**no**, 1
aba	**yes**, 0
b	**yes**, 0
aa	**no**, $\perp$
abb	**no**, $\perp$
$abaa$	**yes**, 0
$abab$	**yes**, 0

	ε
ε	**no**, 0
a	**no**, 1
ab	**no**, 1
aba	**yes**, 0
abb	**no**, 1
b	**yes**, 0
aa	**no**, $\perp$
$abaa$	**yes**, 0
$abab$	**yes**, 0
$abba$	**yes**, 0
$abbb$	**no**, $\perp$

	ε
ε	**no**, 0
a	**no**, 1
ab	**no**, 1
aba	**yes**, 0
abb	**no**, 1
$abbb$	**no**, 1
b	**yes**, 0
aa	**no**, $\perp$
$abaa$	**yes**, 0
$abab$	**yes**, 0
$abba$	**yes**, 0
$abbba$	**yes**, 0
$abbbb$	**no**, $\perp$

Figure 5.7: Observation tables exposing an infinite loop when using the L^* algorithm.

obtain the second table of Figure 5.7.
Notice that the row for abb is now **no**, 1. That is, the row changed, as the set *Pref* $(\mathcal{O}_{\leq 1})$ now contains abb as a prefix of $abba \in L_{\leq 1}$. Again, the table is not closed due to $abbb$, meaning that $abbb$ must be moved to the upper part of the table. We obtain the third table of Figure 5.7 and exactly the same problem: the row for $abbb$ changed and the table is not closed due to $abbbb$, and so on. Hence, trying to make the table closed leads to an infinite loop. We thus need to refine L^* to take into account two different sets of separators: one that is used to know whether a word is in the target language, and one for the counter values.

5.4.3. Approximation sets

To avoid an infinite loop when making the table closed, as described in the previous example, we modify both the concept of table and how to derive an equivalence relation from that table. As already explained, we introduce the set $\widehat{S}$. However, this means that it is harder to extract a relation from $\mathcal{O}_{\leq \ell}$ that approximates $\sim_{\mathcal{A}}$ (up to the counter limit ℓ). In particular, the presence of the $\perp$ symbols may lead to cases where two words $u \sim_{\mathcal{A}} v$ have different row contents, *i.e.*, $C(us) \neq C(vs)$ for some $s \in S$. We thus need to treat $\perp$ as a sort of wildcard, and require equality of rows, up to the $\perp$ cells. Interestingly, such wildcard entries in observation tables also feature in the work of Leucker and Neider on learning from an "inexperienced" teacher [LN12].

[LN12]: Leucker et al. (2012), "Learning Minimal Deterministic Automata from Inexperienced Teachers"

Definition 5.4.9 (Approximation set)**.** Let $\mathcal{O}_{\leq \ell}$ be an observation table up to ℓ, and $u, v \in R \cup R\Sigma$. Then, $u \in$ *Approx*(v) if and only if:

- for all $s \in S$, $T(us) = T(vs)$, and
- for all $s \in S$, if $C(us) \neq \perp$ and $C(vs) \neq \perp$, then $C(us) = C(vs)$.

The set *Approx*(v) is called an *approximation set*.

We highlight that we still require equality of rows for T, *i. e.*, only the values C may differ. Moreover, only the separators in S are considered.

When defining properties over approximation sets, we will mostly focus on words for which we know their counter value. That is, we ignore words that, as far as know with the current knowledge, belong to the bin class of $BG(\mathcal{A})$. Those words are such that $C(u) = \bot$ (which is the only possible value when $u \notin \mathit{Pref}(\mathcal{O}_{\leq \ell})$).

Definition 5.4.10 ($\bot$-word)**.** If $C(u) = \bot$ with $u \in R \cup R\Sigma$, we say that u is a $\bot$-*word*. We write $\overline{R}$ (resp. $\overline{R \cup R\Sigma}$) the set of representatives that are not $\bot$-words:

$$\overline{R} = \{w \in R \mid C(u) \neq \bot\}$$
$$\overline{R \cup R\Sigma} = \{w \in R \cup R\Sigma \mid C(u) \neq \bot\}.$$

Just like in [LN12], a crucial part of our learning algorithm concerns how to obtain an equivalence relation from an observation table with wildcards. Indeed, note that *Approx* does not define an equivalence relation as it is not transitive, *i. e.*, it is not true in general that if $u \in \mathit{Approx}(v)$ and $v \in \mathit{Approx}(w)$, then $u \in \mathit{Approx}(w)$ due to the $\bot$ values. However, *Approx* is reflexive (*i. e.*, $u \in \mathit{Approx}(u)$) and symmetric (*i. e.*, $u \in \mathit{Approx}(v) \Leftrightarrow v \in \mathit{Approx}(u)$).

Example 5.4.11. Let $\mathcal{O}_{\leq \ell}$ be the table from Figure 5.6 (which is repeated in the margin). Notice that $S = \{\varepsilon\}$ and we do not take into account $\hat{S} = \{a, ba\}$ to compute the approximation sets.
We compute $\mathit{Approx}(\varepsilon)$. We can see that $aba \notin \mathit{Approx}(\varepsilon)$ as $T(aba) = \textbf{yes}$ and $T(\varepsilon) = \textbf{no}$. Moreover, $a \notin \mathit{Approx}(\varepsilon)$ since $C(a) \neq \bot, C(\varepsilon) \neq \bot$, and $C(a) \neq C(\varepsilon)$ With the same arguments, we also discard $ab, b, abb, abaa, abab$. Thus, $\mathit{Approx}(\varepsilon) = \{\varepsilon, aa, aaa, aab\}$.
Finally, $\mathit{Approx}(aa) = \{\varepsilon, a, ab, aa, abb, aaa, aab\}$, *i. e.*, every word u such that $T(u) = \textbf{no}$ is in the approximation set of aa, due to the fact that aa is a $\bot$-word.

	ε	a	ba
ε	**no**, 0	**no**	**yes**
a	**no**, 1	**no**	**yes**
ab	**no**, 1	**yes**	**yes**
aba	**yes**, 0	**yes**	**yes**
aa	**no**, $\bot$	**no**	**no**
b	**yes**, 0	**yes**	**yes**
abb	**no**, 1	**yes**	**yes**
$abaa$	**yes**, 0	**yes**	**yes**
$abab$	**yes**, 0	**yes**	**yes**
aaa	**no**, $\bot$	**no**	**no**
aab	**no**, $\bot$	**no**	**no**

We claim that, if we extend the observation table and observe that u is in the approximation set of v, then it must be that u was already in the approximation set of v for every u and v present in the table before the extension. That is, the approximation sets restricted to the already known words cannot increase.[5] We denote by Approx' the approximation sets with respect to the table $\mathcal{O}'_{\leq \ell}$.

5: Of course, the sets may increase in size due to new words.

Lemma 5.4.12. *Let $\mathcal{O}_{\leq \ell}$ and $\mathcal{O}'_{\leq \ell}$ be two observation tables up to the same counter limit $\ell \in \mathbb{N}$ such that $R \cup R\Sigma \subseteq R' \cup R'\Sigma, S \subseteq S'$, and $\hat{S} \subseteq \hat{S}'$. Then,*

$$\forall u, v \in R \cup R\Sigma : u \in \mathit{Approx}'(v) \Rightarrow u \in \mathit{Approx}(v).$$

Proof. By hypothesis, since both tables use the same counter limit ℓ, the same language $L_{\leq \ell}$ is considered. Thus it holds that $T(us) = T'(us)$ for all $u \in R \cup R\Sigma$ and $s \in S$. Moreover, we have $\mathit{Pref}(\mathcal{O}_{\leq \ell}) \subseteq \mathit{Pref}(\mathcal{O}'_{\leq \ell})$ and

$$\forall us \in (R \cup R\Sigma) \cdot S : C(us) \neq \bot \Rightarrow C'(us) = C(us),$$

and some $C(us) = \bot$ can possibly be replaced by $C'(us) \in \{0, \dots, \ell\}$.
Let $u, v \in R \cup R\Sigma$ be such that $u \in \mathit{Approx}'(v)$, *i. e.*,

$$\forall s \in S' : T'(us) = T'(vs)$$

and

$$\forall s \in S' : C'(us) \neq \bot \wedge C'(vs) \neq \bot \Rightarrow C'(us) = C'(vs).$$

We want to show that $u \in \mathit{Approx}(v)$.
Let $s \in S$. We start by proving that $T(us) = T(vs)$. Since $u \in \mathit{Approx}'(v)$, we have $T'(us) = T'(vs)$. Therefore,

$$T(us) = T'(us) = T'(vs) = T(vs).$$

We now show that

$$C(us) \neq \bot \wedge C(vs) \neq \bot \Rightarrow C(us) = C(vs).$$

From $C(us) \neq \bot \wedge C(vs) \neq \bot$, it follows that $C'(us) \neq \bot \wedge C'(vs) \neq \bot$. Since $u \in \mathit{Approx}'(v)$, we have $C(us) = C'(us) = C'(vs) = C(vs)$. □

We now move towards formalizing the relation between $\sim_{\mathcal{A}}$ and *Approx*. Namely, we can show that *Approx* is coarser than $\sim_{\mathcal{A}}$ when we consider (extended) representatives that are not $\bot$-words. This restriction to $\overline{R \cup R\Sigma}$ allows us to reason only on words that are known to be in the prefix of the target language. We prove that two such words equivalent according to $\sim_{\mathcal{A}}$ must necessarily have the same row contents. Furthermore, all $\bot$-words are in the same approximation set (which represents the "bin class" of $\sim_{\mathcal{A}}$), which follows naturally from the definition of a $\bot$-word. The proper proof is deferred to Section A.3.

Proposition 5.4.13. *Let $\mathcal{O}_{\leq \ell}$ be an observation table up to $\ell \in \mathbb{N}$. Then,*

$$\forall u, v \in \overline{R \cup R\Sigma} : u \sim_{\mathcal{A}} v \Rightarrow u \in \mathit{Approx}(v).$$
$$\forall u, v \in R \cup R\Sigma \setminus \overline{R \cup R\Sigma} : \mathit{Approx}(u) = \mathit{Approx}(v).$$

Towards a congruence

In order to be able to construct a hypothesis from $\mathcal{O}_{\leq \ell}$, we have to derive a congruence from the table. As is done in L^* [Ang87], we define some constraints that the table must respect in order to obtain a congruence relation from *Approx*. This is more complex than for L^*, namely, the table must be closed, Σ-consistent, and $\bot$-consistent. The first two constraints are similar to the ones already imposed by L^* while the last one is new. Crucially, it implies that *Approx* is transitive.[6]

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

6: That is, under this condition, *Approx* becomes an equivalence relation.

Definition 5.4.14 (Closed, Σ-consistent, and $\bot$-consistent table). Let $\mathcal{O}_{\leq \ell}$ be an observation table up to $\ell \in \mathbb{N}$. We say the table is:

- *closed* if $\mathit{Approx}(u) \cap R \neq \emptyset$ for all $u \in R\Sigma$, and *open* otherwise,
- Σ*-consistent* if for all $u \in R, a \in \Sigma$, and $v \in \mathit{Approx}(u) \cap R$, $u \cdot a \in$

Approx$(v \cdot a)$, and Σ-*inconsistent* otherwise,

- $\perp$-*consistent* if for all $u, v \in R \cup R\Sigma$ such that $u \in$ *Approx*(v), it holds that for all $s \in S$ $C(u \cdot s) = \perp$ if and only if $C(v \cdot s) = \perp$, and $\perp$-*inconsistent* otherwise.

Example 5.4.15. Let $\mathcal{O}_{\leq \ell}$ be the table from Figure 5.6 (which is repeated in the margin). We have *Approx*$(b) \cap R \neq \emptyset$ because $aba \in$ *Approx*(b). More generally one can check that $\mathcal{O}_{\leq \ell}$ is closed.
However, $\mathcal{O}_{\leq \ell}$ is not Σ-consistent. Indeed,

$$\varepsilon \cdot b \notin \bigcap_{v \in \mathit{Approx}(\varepsilon) \cap R} \mathit{Approx}(v \cdot b)$$

since *Approx*$(\varepsilon) \cap R = \{\varepsilon, aa\}$ and $\varepsilon \cdot b \notin$ *Approx*$(aa \cdot b)$.
Finally, $\mathcal{O}_{\leq \ell}$ is also not $\perp$-consistent since $aa \in$ *Approx*(ε) but $C(aa) = \perp$ and $C(\varepsilon) = 0$.

	ε	a	ba
ε	**no**, 0	**no**	**yes**
a	**no**, 1	**no**	**yes**
ab	**no**, 1	**yes**	**yes**
aba	**yes**, 0	**yes**	**yes**
aa	**no**, $\perp$	**no**	**no**
b	**yes**, 0	**yes**	**yes**
abb	**no**, 1	**yes**	**yes**
$abaa$	**yes**, 0	**yes**	**yes**
$abab$	**yes**, 0	**yes**	**yes**
aaa	**no**, $\perp$	**no**	**no**
aab	**no**, $\perp$	**no**	**no**

Before explaining how to ensure that the table satisfies the three conditions, let us argue why it is sufficient to obtain a right-congruence. We highlight again that *Approx* is transitive when $\mathcal{O}_{\leq \ell}$ is $\perp$-consistent. This allows us to define an equivalence relation similarly to what is done in L^*: two rows are equivalent when they have the same contents. Section 5.4.4 explains how to construct an ROCA from this relation.

Definition 5.4.16 (Relation over R). Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ-, and $\perp$-consistent observation table up to ℓ. We say that two words $u, v \in R \cup R\Sigma$ are $\equiv_{\mathcal{O}_{\leq \ell}}$-equivalent, denoted by $u \equiv_{\mathcal{O}_{\leq \ell}} v$, if and only if $u \in$ *Approx*(v).

It is not hard to see that $\equiv_{\mathcal{O}_{\leq \ell}}$ is reflexive and symmetric. Showing that it is transitive requires more care, due to the approximation sets, but can be obtained from the definition of a $\perp$-consistent table. Finally, as the next proposition states, $\equiv_{\mathcal{O}_{\leq \ell}}$ is a right-congruence over R when the table is closed, Σ- and $\perp$-consistent.

Proposition 5.4.17. *Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell \in \mathbb{N}$. Then, $\equiv_{\mathcal{O}_{\leq \ell}}$ is an equivalence relation over $R \cup R\Sigma$ that is a congruence over R.*

Proof. We first show that $\equiv_{\mathcal{O}_{\leq \ell}}$ is an equivalence relation over $R \cup R\Sigma$, before proving that it is a congruence over R.

$\equiv_{\mathcal{O}_{\leq \ell}}$ is an equivalence relation over $R \cup R\Sigma$. It is easy to see that $\equiv_{\mathcal{O}_{\leq \ell}}$ is reflexive and symmetric. We thus need to show the transitivity aspect. Let $u, v, w \in R \cup R\Sigma$ be such that $u \in$ *Approx*(v) and $v \in$ *Approx*(w). We want to show that $u \in$ *Approx*(w), *i.e.*,

$$\forall s \in S : T(us) = T(ws)$$

and

$$\forall s \in S : C(us) \neq \perp \wedge C(ws) \neq \perp \Rightarrow C(us) = C(ws).$$

Let $s \in S$. By hypothesis, we have $T(us) = T(vs)$ and $T(vs) = T(ws)$. So, $T(us) = T(ws)$.
Now, assume $C(us) \neq \bot \wedge C(ws) \neq \bot$. Since the table is $\bot$-consistent and $u \in \mathit{Approx}(v)$, it must hold that $C(vs) \neq \bot$. Therefore, $C(us) = C(vs)$. Likewise, we deduce that $C(vs) = C(ws)$ since $v \in \mathit{Approx}(w)$. We conclude that $C(us) = C(ws)$.

$\equiv_{\mathcal{O}_{\leq \ell}}$ is a congruence over R. Second, we prove that $\equiv_{\mathcal{O}_{\leq \ell}}$ is a congruence over R. Let $u, v \in R$ be such that $u \equiv_{\mathcal{O}_{\leq \ell}} v$ and let $a \in \Sigma$. Since the table is closed, there exist $u', v' \in R$ such that $u' \equiv_{\mathcal{O}_{\leq \ell}} ua$ and $v' \equiv_{\mathcal{O}_{\leq \ell}} va$. Since the table is Σ-consistent, we have

$$ua \in \bigcap_{w \in \mathit{Approx}(u) \cap R} \mathit{Approx}(wa).$$

Thus, as $v \in \mathit{Approx}(u) \cap R$, we have $ua \in \mathit{Approx}(va)$, *i. e.*, $ua \equiv_{\mathcal{O}_{\leq \ell}} va$. It follows that $u' \equiv_{\mathcal{O}_{\leq \ell}} v'$. □

Let us now explain how one can extend the table in order to make sure it satisfies the three conditions. While we state lemmas giving the growth of the table after resolving one problem, showing the following proposition saying that we can always obtain a closed, Σ- and $\bot$-consistent table in finite time requires more work. The proof is deferred to Section A.8.2.

Proposition 5.4.18. *Given an observation table $\mathcal{O}_{\leq \ell}$ up to $\ell \in \mathbb{N}$, there exists an algorithm that makes it closed, Σ- and $\bot$-consistent in a finite amount of time.*

Making the table closed

Assume $\mathcal{O}_{\leq \ell}$ is open, *i. e.*,

$$\exists u \in R\Sigma : \mathit{Approx}(u) \cap R = \emptyset.$$

We say that we have a *u-openness*. Notice that we have $u \notin R$ since $u \in \mathit{Approx}(u)$ and $\mathit{Approx}(u) \cap R = \emptyset$. We thus add u as a new representative.[7] Then, u is no longer a word making the table open, *i. e.*, the previous u-openness is resolved.

7: $S, \widehat{S}$ are left unchanged.

The next lemma states that this construction is correct, in the sense that the openness is resolved, and the number of queries to do so. The proof is given in Section A.4.

Lemma 5.4.19. *Let $\mathcal{O}_{\leq \ell} = (R, S, \hat{S}, T, C)$ be an observation table and $\mathcal{O}'_{\leq \ell} = (R', S', \hat{S}', T', C')$ be the observation table obtained after resolving a u-openness (with $u \in R\Sigma \setminus R$). Then,*

- $|R'| = |R| + 1$, $|S'| = |S|$, $\left|\hat{S}'\right| = \left|\hat{S}\right|$,
- *$\mathit{Approx}'(u) \cap R' = \{u\}$, and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in $|\mathcal{O}_{\leq \ell}|$.*

Making the table Σ-consistent

Assume $\mathcal{O}_{\leq \ell}$ is Σ-inconsistent, *i. e.*,

$$\exists ua \in R\Sigma, \exists v \in R : v \in \mathit{Approx}(u) \text{ and } ua \notin \mathit{Approx}(va).$$

We say that we have a (u, v, a)-Σ-*inconsistency*. By definition, there exists $s \in S$ such that either

- $T(uas) \neq T(vas)$, or
- $C(uas) \neq \bot, C(vas) \neq \bot$, and $C(uas) \neq C(vas)$.

In both cases, we add as in both S and $\widehat{S}$.[8]

8: R is left unchanged. We add as to $\widehat{S}$ in order to maintain $S' \subseteq \widehat{S}'$.

The next lemma indicates that the (u, v, a)-Σ-inconsistency is resolved since it states that v no longer belongs to $\mathit{Approx}'(u)$. Moreover, this approximation set gets smaller, and the complexity in number of queries is also provided. The proof is deferred to Section A.5.

Lemma 5.4.20. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and $\mathcal{O}'_{\leq \ell}$ be the observation table obtained after resolving a (u, v, a)-Σ-inconsistency (with $ua \in R\Sigma$ and $v \in R$). Then,*

- $|R'| = |R|, |S'| = |S| + 1, \left|\widehat{S}'\right| = \left|\widehat{S}\right| + 1,$
- $v \notin \mathit{Approx}'(u)$,
- $\left|\mathit{Approx}'(u)\right| < |\mathit{Approx}(u)|$, *and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in* $|\mathcal{O}_{\leq \ell}|$.

Making the table $\bot$-consistent

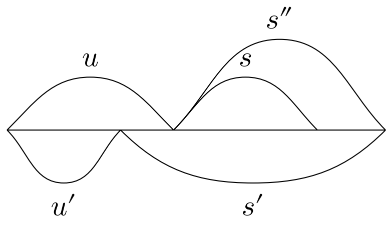
Assume $\mathcal{O}_{\leq \ell}$ is $\bot$-inconsistent, *i. e.*, there exist $u, v \in R \cup R\Sigma$ and $s \in S$ such that

$$u \in \mathit{Approx}(v) \wedge (C(us) \neq \bot \Leftrightarrow C(vs) = \bot).$$

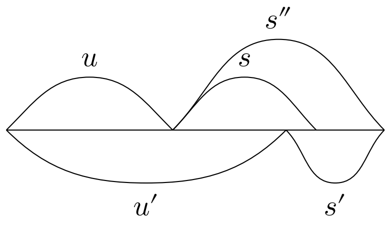
We say that we have a (u, v, s)-$\bot$-*inconsistency* and we call *mismatch* the disequality $C(us) \neq \bot \Leftrightarrow C(vs) = \bot$. We now explain how to resolve this $\bot$-inconsistency. Let us assume, without loss of generality, that $C(us) \neq \bot$ and $C(vs) = \bot$. So, $us \in \mathit{Pref}(\mathcal{O}_{\leq \ell})$, *i. e.*, there exist $u' \in R \cup R\Sigma$ and $s' \in \widehat{S}$ such that $us \in \mathit{Pref}(u's')$ and $T(u's') = \textbf{yes}$. We denote by s'' the word such that $us'' = u's'$. Notice that s is a prefix of s''. We have two cases according to whether u' is a prefix of u (see Figure 5.8a) or u is a proper prefix of u' (see Figure 5.8b).

- We first suppose that u' is a prefix of u.
  Let us show that $s'' \in \widehat{S} \setminus S$. As $\widehat{S}$ is suffix-closed, and s'' is a suffix of $s' \in \widehat{S}$, it follows that s'' belongs to $\widehat{S}$. By contradiction, assume $s'' \in S$. Since $u \in \mathit{Approx}(v)$ and $T(us'') = T(u's') = \textbf{yes}$, we have $T(vs'') = T(us'') = \textbf{yes}$. Moreover, since s is a prefix of s'', it holds that $vs \in \mathit{Pref}(\mathcal{O}_{\leq \ell})$. This means that $C(vs) \neq \bot$ which is a contradiction.
  To resolve the (u, v, s)-$\bot$-inconsistency, we add all suffixes of s'' in S.[9]

9: R and $\widehat{S}$ are left unchanged.



(a) u' is a prefix of u.



(b) u is a proper prefix of u'.

Figure 5.8: The different words used to resolve a (u, v, s)-$\perp$-inconsistency.

▶ We then suppose that u is a proper prefix of u'.
Then, to resolve the (u, v, s)-$\perp$-inconsistency, we have two cases:
   - If $vs'' \in L_{\leq \ell}$, we add all suffixes of s'' in $\widehat{S}$.[10]
   - If $vs'' \notin L_{\leq \ell}$, we add all suffixes of s'' in $\widehat{S}$ and in S.[11]

10: R and S are left unchanged.

11: R is left unchanged.

The next lemma indicates that the (u, v, s)-$\perp$-inconsistency is indeed resolved, *i.e.*, $u \notin \mathit{Approx}'(v)$ or the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated. It also gives the complexity in number of queries. The proof is deferred to Section A.6.

Lemma 5.4.21. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and $\mathcal{O}'_{\leq \ell}$ be the observation table obtained after resolving a (u, v, s)-$\perp$-inconsistency (with $u, v \in R \cup R\Sigma$ and $s \in S$). Then,*

▶ $|R'| = |R|$, $|S'| \leq |R \cup R\Sigma| + \left|\widehat{S}\right|$, $\left|\widehat{S}'\right| \leq |R \cup R\Sigma| + \left|\widehat{S}\right|$,
▶ *if u' is a prefix of u, then $u \notin \mathit{Approx}'(v)$,*
▶ *if u is a proper prefix of u', then,*
   - *if $vs'' \in L_{\leq \ell}$, then $u \in \mathit{Approx}'(v)$ implies that $C'(us) = C'(vs)$, and*
   - *if $vs'' \notin L_{\leq \ell}$, then $u \notin \mathit{Approx}'(v)$,*

▶ *either $\left|\mathit{Approx}'(v)\right| < |\mathit{Approx}(v)|$ or the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated,*
▶ *if the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated, we have $|S'| = |S|$, and*
▶ *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in $|\mathcal{O}_{\leq \ell}|$.*

5.4.4. Hypothesis construction

Let us now explain how to construct the hypothesis DFA from $\mathcal{O}_{\leq \ell}$, once the table is closed, Σ- and $\perp$-consistent. The equivalence classes of $\equiv_{\mathcal{O}_{\leq \ell}}$ form its set of states and transitions are easily defined thanks to Proposition 5.4.17. That is, the construction of the DFA follows naturally from the fact that $\equiv_{\mathcal{O}_{\leq \ell}}$ is a right-congruence. For convenience, we give here a complete construction.

Proposition 5.4.17. Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell \in \mathbb{N}$. Then, $\equiv_{\mathcal{O}_{\leq \ell}}$ is an equivalence relation over $R \cup R\Sigma$ that is a congruence over R.

Definition 5.4.22 (Hypothesis construction)**.** Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell \in \mathbb{N}$. From $\equiv_{\mathcal{O}_{\leq \ell}}$, we define the DFA $\mathcal{H}_{\leq \ell} = (\Sigma, Q, q_0, F, \delta)$ with:

- $Q = \{[\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}} \mid u \in R\}$,
- $q_0 = [\![\varepsilon]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}}$,
- $F = \{[\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}} \mid T(u) = \textbf{yes}\}$, and
- the (total) transition function δ is defined such that for all $[\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}} \in Q$ and $a \in \Sigma$,
$$[\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}} \xrightarrow{a} [\![ua]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}} \in \mathit{runs}(\mathcal{H}_{\leq \ell}).$$

The next lemma states that $\mathcal{H}_{\leq \ell}$ is well-defined and its definition makes sense with regards to the information stored in the table $\mathcal{O}_{\leq \ell}$.

Lemma 5.4.23. *Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell \in \mathbb{N}$, and $\mathcal{H}_{\leq \ell}$ be the constructed DFA. Then,*

- *$\mathcal{H}_{\leq \ell}$ is well-defined and*
- *for all $u \in R \cup R\Sigma : u \in \mathcal{L}(\mathcal{H}_{\leq \ell}) \Leftrightarrow u \in L_{\leq \ell}$.*

Proof. Let us first explain why $\mathcal{H}_{\leq \ell}$ is well-defined. The initial state is well defined since $\varepsilon \in R$. The set of final states is well defined because

$$u \equiv_{\mathcal{O}_{\leq \ell}} v \wedge T(u) = \textbf{yes} \Rightarrow T(v) = \textbf{yes}.$$

The transition function is well defined because the table $\mathcal{O}_{\leq \ell}$ is closed and $\equiv_{\mathcal{O}_{\leq \ell}}$ is a congruence.
Let us prove the second statement of the lemma. An easy induction shows that
$$\forall u \in R \cup R\Sigma : q_0 \xrightarrow{u} [\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}}.$$
Therefore, for all such u, we get $u \in \mathcal{L}(\mathcal{H}_{\leq \ell}) \Leftrightarrow u \in L_{\leq \ell}$. □

Isomorphism with the behavior graph

Finally, let us argue that it is reasonable to learn $L_{\leq \ell}$. That is, once ℓ is big enough, we want to be able to construct an ROCA that accepts the target language L. If $\mathcal{H}_{\leq \ell}$ accepts the language $L_{\leq \ell}$, the next proposition states that the initial fragments (up to a certain counter limit) of both $\mathcal{H}_{\leq \ell}$ and $BG(\mathcal{A})$ are isomorphic. This means that, if ℓ is big enough, we can extract a periodic description from $\mathcal{H}_{\leq \ell}$ that is valid for $BG(\mathcal{A})$. A proof is given in Section A.7.

Proposition 5.4.24. *Let $BG(\mathcal{A})$ be the behavior graph of an ROCA $\mathcal{A}$, K be its width, m, k be the offset and the period of a periodic description of $BG(\mathcal{A})$, $s = m + (K \cdot k)^4$, and $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell > s$ such that $\mathcal{L}(\mathcal{H}_{\leq \ell}) = L_{\leq \ell}$. Then, the subgraphs of $BG(\mathcal{A})$ and $\mathcal{H}_{\leq \ell}$ restricted to the reachable and co-reachable states and to the levels 0 to $\ell - s$ are isomorphic.*

Algorithm 5.1: Overall L^*_{ROCA}algorithm.

```
1:  Initialize $\mathcal{O}_{\leq \ell}$ with $\ell = 0, R = S = \hat{S} = \{\varepsilon\}$
2:  while true do
3:      Make $\mathcal{O}_{\leq \ell}$ closed, $\Sigma$-consistent, and $\perp$-consistent
4:      Construct the DFA $\mathcal{H}_{\leq \ell}$ from $\mathcal{O}_{\leq \ell}$
5:      $v \leftarrow$ PEQ$(\mathcal{H}_{\leq \ell}, \ell)$
6:      if $v \neq$ yes then Update $\mathcal{O}_{\leq \ell}$ with $v$          ▷ $\ell$ is not modified
7:      else
8:          $W \leftarrow \emptyset$
9:          for all periodic descriptions $\alpha$ of $\mathcal{H}_{\leq \ell}$ do
10:             Construct the VOCA $\mathcal{H}_\alpha$ from $\alpha$
11:             $v_\alpha \leftarrow$ EQ$(\mathcal{H}_\alpha)$
12:             if $v_\alpha =$ yes then return $\mathcal{H}_\alpha$
13:             else if $v_\alpha \in \mathcal{L}(\mathcal{H}_\alpha) \Leftrightarrow v_\alpha \in \mathcal{L}(\mathcal{H}_{\leq \ell}) \land$ MQ$(v_\alpha) =$ yes then
14:                 $W \leftarrow W \cup \{v_\alpha\}$
15:         if $W = \emptyset$ then
16:             Let $\mathcal{H}^{\text{ROCA}}_{\leq \ell}$ be $\mathcal{H}_{\leq \ell}$ seen as an ROCA
17:             $w \leftarrow$ EQ$(\mathcal{H}^{\text{ROCA}}_{\leq \ell})$
18:             if $w =$ yes then return $\mathcal{H}^{\text{ROCA}}_{\leq \ell}$          ▷ The target language is regular
19:             else $W \leftarrow W \cup \{w\}$
20:         Select an arbitrary $w$ from $W$
21:         $\ell \leftarrow \max_{x \in Pref(w)}$ CVQ$(x)$          ▷ $\ell$ is increased
22:         Update $\mathcal{O}_{\leq \ell}$ with $w$
```

5.4.5. Main loop

We are now ready to give the main loop of L^*_{ROCA}. Algorithm 5.1 gives a pseudo-code. We start by initializing the observation table $\mathcal{O}_{\leq \ell}$ with $\ell = 0, R = S = \hat{S} = \{\varepsilon\}$. Then, we make the table closed, Σ-, and $\perp$-consistent, construct the DFA $\mathcal{H}_{\leq \ell}$, and ask **PEQ**$(\mathcal{H}_{\leq \ell})$. If the teacher answers positively, we have learned a DFA accepting $L_{\leq \ell}$. Otherwise, we use the provided counterexample to update the table without increasing ℓ.[12]

12: Counterexample processing is given below.

Once the learned DFA $\mathcal{H}_{\leq \ell}$ accepts the language $L_{\leq \ell}$, we know by Proposition 5.4.24 that the first levels of $\mathcal{H}_{\leq \ell}$ and $BG(\mathcal{A})$ are isomorphic (once restricted to the reachable and co-reachable states). This means that, once we have learned a long enough initial fragment, we can extract a periodic description from $\mathcal{H}_{\leq \ell}$ that is valid for $BG(\mathcal{A})$. Hence we extract all possible periodic descriptions α from $\mathcal{H}_{\leq \ell}$, exactly as for VOCAs. By Proposition 5.3.12, each of these descriptions α yields an ROCA $\mathcal{H}_\alpha$ on which we ask an equivalence query. If the teacher answers positively, we have learned an ROCA accepting L and we are done. Otherwise, the teacher returns a counterexample v_α.

Proposition 5.4.24. Let $BG(\mathcal{A})$ be the behavior graph of an ROCA $\mathcal{A}$, K be its width, m, k be the offset and the period of a periodic description of $BG(\mathcal{A})$, $s = m + (K \cdot k)^4$, and $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell > s$ such that $\mathcal{L}(\mathcal{H}_{\leq \ell}) = L_{\leq \ell}$. Then, the subgraphs of $BG(\mathcal{A})$ and $\mathcal{H}_{\leq \ell}$ restricted to the reachable and co-reachable states and to the levels 0 to $\ell - s$ are isomorphic.

Given the fact that we will have to increase ℓ if none of the ROCAs accepts L, we must make sure that we can call **CVQ**(v_α), *i.e.*, that v_α is in the prefix of the target language. First, we check whether $w \notin \mathcal{L}(\mathcal{H}_\alpha) \Leftrightarrow w \in \mathcal{L}(\mathcal{H}_{\leq \ell})$, which may happen if the description α only considers the first few levels of the DFA $\mathcal{H}_{\leq \ell}$ and, by doing so, discards important knowledge appearing on the further levels. If it does not hold, then we know that every word in $Pref(\mathcal{O}_{\leq \ell})$

is correctly accepted or rejected by $\mathcal{H}_\alpha$. As we are not able to guess whether v_α is in $\mathit{Pref}(L)$ from the current knowledge, we simply check whether $v_\alpha \in L$. If it is the case, we know that we can call **CVQ**(v_α), as required.

If we do not have any ROCAs or every counterexample was discarded, we use $\mathcal{H}_{\leq \ell}$ directly as an ROCA (this can be done simply by constructing an ROCA that simulates $\mathcal{H}_{\leq \ell}$ without ever modifying the counter). Since we know that $\mathcal{L}(\mathcal{H}_{\leq \ell}) = L_{\leq \ell}$, we are sure the returned counterexample is usable. Note that if L is regular, it may happen that $\mathcal{H}_{\leq \ell}$ accepts L. In which case, we simply return the ROCA constructed from $\mathcal{H}_{\leq \ell}$.

Proposition 5.3.12. Let $BG(\mathcal{A})$ be the behavior graph of some sound ROCA $\mathcal{A}$ and $\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega$ be an ultimately periodic description of $BG(\mathcal{A})$ with offset m and period k. Then, one can construct an ROCA $\mathcal{A}_\alpha$ from α such that

- $\mathcal{L}(\mathcal{A}_\alpha) = \mathcal{L}(\mathcal{A})$, and
- the size of $\mathcal{A}_\alpha$ is polynomial in m, k and $\mathit{width}(BG(\mathcal{A}))$.

Finally, let w be an arbitrary counterexample among the ones that were not discarded. The new counter limit is the height of w, *i.e.*, $\ell = \mathit{height}^{\mathcal{A}}(w)$.[13] This is possible thanks to the above constraints over the counterexamples. We then extend and update the table by adding $\mathit{Pref}(w)$ to R. Notice that this operation may change some values of T since a word that was rejected may now be accepted due to the higher counter limit.

13: This requires a counter value query.

Handling counterexamples

Given an observation table $\mathcal{O}_{\leq \ell}$ that is closed, Σ-consistent and $\perp$-consistent, let $\mathcal{H}_{\leq \ell}$ be the DFA constructed from $\mathcal{O}_{\leq \ell}$ like described in Definition 5.4.22. If the teacher's answer to a **PEQ** over $\mathcal{H}_{\leq \ell}$ is positive, the learned DFA $\mathcal{H}_{\leq \ell}$ exactly accepts $L_{\leq \ell}$. Otherwise, the teacher returns a counterexample, that is, a word $w \in \Sigma^*$ such that

$$w \in L_{\leq \ell} \Leftrightarrow w \notin \mathcal{L}(\mathcal{H}_{\leq \ell}).$$

We extend and update the table $\mathcal{O}_{\leq \ell}$ to obtain a new observation table $\mathcal{O}'_{\leq \ell} = (R', S', \widehat{S}', T', C')$ where we add every prefix of w as a new representative, *i.e.*,

$$R' = R \cup \mathit{Pref}(w).$$

We then compute T' and C' using membership and counter value queries.[14]

14: Observe that the sets of separators are unchanged so far.

Counterexamples coming from **EQ** are handled in the same way, except that the counter limit is increased to the height of the counterexample. These two procedures allow us to state the growth of the representatives and separators sets, as well as the number of needed queries.

Lemma 5.4.25. *Let $\mathcal{O}_{\leq \ell}$ be an observation table, ζ be the length of the counterexample returned by a* **PEQ** *or* **EQ**, *and $\mathcal{O}'_{\leq \ell'}$ be the new table obtained by processing the counterexample. Then,*

- $|R'| \leq |R| + \zeta$, $|S'| = |S|$, $\left|\widehat{S}'\right| = \left|\widehat{S}\right|$, *and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in ζ and $|\mathcal{O}_{\leq \ell}|$.*

Proof. The first item is easily proved, given how a counterexample is handled. Let us prove the last item. The worst case with respect to the number of queries happens when $\ell' > \ell$. In this case, we potentially have to (re)compute all values in T' and C', as the counter limit has increased. By Lemma 5.4.7,

Lemma 5.4.7. Filling T and C requires a polynomial number of **MQ** and **CVQ** in the sizes of $R \cup R\Sigma$ and $\widehat{S}$.

this requires a number of **MQ** and **CVQ** that is polynomial in ζ and $|\mathcal{O}_{\leq \ell}|$. $\square$

Once the prefixes of the counterexample are added as new representatives, we can resume the usual operations, *i.e.*, making the table closed, Σ- and $\bot$-consistent before constructing the next hypothesis, and so on. We show that, after processing a counterexample from a partial equivalence query, the new equivalence relation $\equiv_{\mathcal{O}'_{\leq \ell}}$ (obtained once the table is stabilized) is a refinement of $\equiv_{\mathcal{O}_{\leq \ell}}$ with strictly more equivalence classes.

Proposition 5.4.26. *Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\bot$-consistent observation table up to $\ell \in \mathbb{N}$, and $\mathcal{O}'_{\leq \ell}$ be the closed, Σ- and $\bot$-consistent observation table obtained after processing a counterexample. Then,*

$$\forall u, v \in R \cup R\Sigma : u \equiv_{\mathcal{O}'_{\leq \ell}} v \Rightarrow u \equiv_{\mathcal{O}_{\leq \ell}} v.$$

Furthermore, the index of $\equiv_{\mathcal{O}'_{\leq \ell}}$ is strictly greater than the index of $\equiv_{\mathcal{O}_{\leq \ell}}$.

Proof. The first part of the claim is an immediate consequence of Lemma 5.4.12.
We focus on showing that the index of $\equiv_{\mathcal{O}'_{\leq \ell}}$ is strictly greater than the index of $\equiv_{\mathcal{O}_{\leq \ell}}$. By contradiction, let us assume this is false. Let $\mathcal{H}_{\leq \ell}$ (resp. $\mathcal{H}'_{\leq \ell}$) be the DFA constructed from $\equiv_{\mathcal{O}_{\leq \ell}}$ (resp. $\equiv_{\mathcal{O}'_{\leq \ell}}$). By definition of the automata and the first part of the proposition, it holds that $\mathcal{H}_{\leq \ell}$ and $\mathcal{H}'_{\leq \ell}$ are isomorphic.
However, w is a counterexample for $\mathcal{H}_{\leq \ell}$, *i.e.*,

$$w \notin L_{\leq \ell} \Leftrightarrow w \in \mathcal{L}(\mathcal{H}_{\leq \ell}).$$

By Lemma 5.4.23 and since $w \in R'$, we have

$$w \in L_{\leq \ell} \Leftrightarrow w \in \mathcal{L}(\mathcal{H}'_{\leq \ell}).$$

This is a contradiction with $\mathcal{H}_{\leq \ell}$ and $\mathcal{H}'_{\leq \ell}$ being isomorphic. $\square$

Lemma 5.4.12. Let $\mathcal{O}_{\leq \ell}$ and $\mathcal{O}'_{\leq \ell}$ be two observation tables up to the same counter limit $\ell \in \mathbb{N}$ such that $R \cup R\Sigma \subsetneq R' \cup R'\Sigma, S \subseteq S'$, and $\hat{S} \subseteq \hat{S}'$. Then, for all $u, v \in R \cup R\Sigma$ such that $u \in \mathit{Approx}'(v)$, we have $u \in \mathit{Approx}(v)$.

Lemma 5.4.23. Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\bot$-consistent observation table up to $\ell \in \mathbb{N}$, and $\mathcal{H}_{\leq \ell}$ be the constructed DFA. Then,

- $\mathcal{H}_{\leq \ell}$ is well-defined and
- for all $u \in R \cup R\Sigma : u \in \mathcal{L}(\mathcal{H}_{\leq \ell}) \Leftrightarrow u \in L_{\leq \ell}$.

From Proposition 5.4.26 and Lemma 5.4.12, we deduce that after a finite number of steps, we will obtain an observation table $\mathcal{O}_{\leq \ell}$ and its corresponding DFA $\mathcal{H}_{\leq \ell}$ such that $\mathcal{L}(\mathcal{H}_{\leq \ell}) = L_{\leq \ell}$. That is, we eventually learn $L_{\leq \ell}$ in finite time.

Corollary 5.4.27. *Let $\ell \in \mathbb{N}$ be a counter limit. With the described learning process, a DFA accepting $L_{\leq \ell}$ is eventually learned.*

Proof. On one hand, given an observation table $\mathcal{O}_{\leq \ell}$ up to ℓ, by Lemma 5.4.12, we have that for all $u, v \in \overline{R \cup R\Sigma}$, $u \sim_{\mathcal{A}} v \Rightarrow u \equiv_{\mathcal{O}_{\leq \ell}} v$, and all the words in $R \cup R\Sigma \setminus \overline{R \cup R\Sigma}$ are in the same equivalence class of $\equiv_{\mathcal{O}_{\leq \ell}}$.
On the other hand, by Lemma 5.4.3, the number of equivalence classes of $\sim_{\mathcal{A}}$ up to counter limit ℓ is bounded by $(\ell + 1)|Q^{\mathcal{A}}| + 1$. With Proposition 5.4.26, it follows that the index of $\equiv_{\mathcal{O}_{\leq \ell}}$ eventually stabilizes.
This means that the teacher stops giving any counterexample and the final

Lemma 5.4.3. The number of states of $BG_{\leq \ell}(\mathcal{A})$ is at most $(\ell + 1) \cdot |Q^{\mathcal{A}}|$.

	ε
ε	**no**, 0
a	**no**, $\bot$
b	**yes**, 0

(a) Initial table.

	ε
ε	**no**, 0
b	**yes**, 0
a	**no**, $\bot$
ba	**yes**, 0
bb	**yes**, 0

(b) After resolving the b-openness.

	ε	b
ε	**no**, 0	**yes**, 0
b	**yes**, 0	**yes**, 0
a	**no**, $\bot$	**no**, $\bot$
ba	**yes**, 0	**yes**, 0
bb	**yes**, 0	**yes**, 0

(c) After resolving the $(\varepsilon, a, \varepsilon)$-$\bot$-inconsistency.

	ε	b
ε	**no**, 0	**yes**, 0
b	**yes**, 0	**yes**, 0
a	**no**, $\bot$	**no**, $\bot$
ba	**yes**, 0	**yes**, 0
bb	**yes**, 0	**yes**, 0
aa	**no**, $\bot$	**no**, $\bot$
ab	**no**, $\bot$	**no**, $\bot$

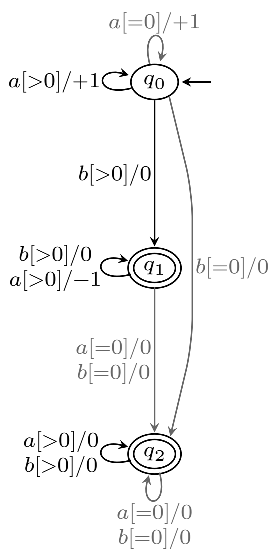
(d) After resolving the a-openness.

Figure 5.9: Observation tables up to zero for Section 5.4.6.

DFA $\mathcal{H}_{\leq \ell}$ accepts $L_{\leq \ell}$. □

5.4.6. Complete example

We conclude the description of our learning algorithm for ROCAs with an example of an execution. Let $\mathcal{A}$ be the ROCA of Figure 5.1 (which is repeated in the margin).



We initialize the observation table with ε as the unique element of both R and $S = \widehat{S}$. Moreover, we set the counter limit to be $\ell = 0$, *i.e.*, we start with $\mathcal{O}_{\leq 0}$. We then fill the table with membership and counter value queries. Figure 5.9a gives the resulting table. Observe that $C(a) = \bot$, as a is not in $\mathit{Pref}(\mathcal{O}_{\leq 0})$.

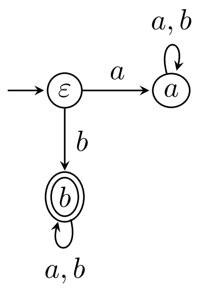
We have the following approximation sets:

$$\mathit{Approx}(\varepsilon) = \mathit{Approx}(a) = \{\varepsilon, a\} \qquad \text{and} \qquad \mathit{Approx}(b) = \{b\}.$$

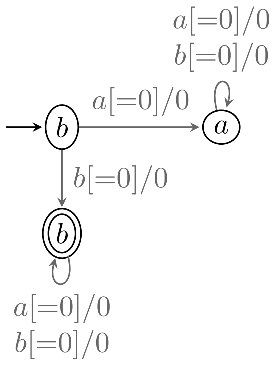
Observe that this table is open, as $\mathit{Approx}(b) \cap R = \emptyset$. We thus add b as a new representative, *i.e.*, R is now $\{\varepsilon, b\}$, and obtain the table of Figure 5.9b with

$$\mathit{Approx}(\varepsilon) = \mathit{Approx}(a) = \{\varepsilon, a\} \qquad \text{and} \qquad \mathit{Approx}(b) = \{b\}.$$

The table is $\bot$-inconsistent, as $a \in \mathit{Approx}(\varepsilon)$ but $C(\varepsilon \cdot \varepsilon) = 0$ and $C(a \cdot \varepsilon) = \bot$. That is, the counter values for the column ε are different. To match with the notations used above when explaining how to make a table $\bot$-consistent, let $u = \varepsilon, v = a, s = \varepsilon, u' = b$, and $s' = \varepsilon$. We have that $u \cdot s = \varepsilon$ is a prefix of $u' \cdot s' = b$, and $T(b) = \textbf{yes}$. Let $s'' = b$ (observe that $u \cdot s'' = b = u' \cdot s'$, as required). Since u is a proper prefix of u', we call $\textbf{MQ}(v \cdot s'') = \textbf{MQ}(a \cdot b)$, which returns **yes**, indicating that $a \cdot b \in \mathcal{L}(\mathcal{A})$. As we need to check whether $a \cdot b \in \mathcal{L}_{\leq 0}(\mathcal{A})$, we also call $\textbf{CVQ}(a \cdot b)$, which returns 1. Hence, $a \cdot b \notin \mathcal{L}_{\leq 0}(\mathcal{A})$.



(a) The hypothesis DFA.



(b) The hypothesis ROCA.

Figure 5.10: The hypotheses DFA and ROCA constructed from the table of Figure 5.9d.

As explained in Section 5.4.3, we add all the suffixes of b to both S and $\hat{S}$. We obtain the table given in Figure 5.9c with

$$\begin{aligned} \mathit{Approx}(\varepsilon) &= \{\varepsilon\}, \\ \mathit{Approx}(b) = \mathit{Approx}(ba) = \mathit{Approx}(bb) &= \{b, ba, bb\}, \end{aligned}$$

and

$$\mathit{Approx}(a) = \{a\}.$$

The table is open, due to a. As before, we then add a as a new representative, which results in the table of Figure 5.9d with

$$\begin{aligned} \mathit{Approx}(\varepsilon) &= \{\varepsilon\}, \\ \mathit{Approx}(b) = \mathit{Approx}(ba) = \mathit{Approx}(bb) &= \{b, ba, bb\}, \end{aligned}$$

and

$$\mathit{Approx}(a) = \mathit{Approx}(aa) = \mathit{Approx}(ab) = \{a, aa, ab\}.$$

Clearly, the table is closed and $\bot$-consistent. It is also not hard to see that it is Σ-consistent, as the intersection of each approximation set with R is a singleton. Hence, we can compute an equivalence relation $\equiv_{\mathcal{O}_{\leq 0}}$ from the table (see Definition 5.4.16) such that:

Definition 5.4.16. Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ-, and $\bot$-consistent observation table up to ℓ. We say that two words $u, v \in R \cup R\Sigma$ are $\equiv_{\mathcal{O}_{\leq \ell}}$-equivalent, denoted by $u \equiv_{\mathcal{O}_{\leq \ell}} v$, if and only if $u \in \mathit{Approx}(v)$.

$$\begin{aligned} [\![\varepsilon]\!]_{\equiv_{\mathcal{O}_{\leq 0}}} &= \{\varepsilon\} \\ [\![b]\!]_{\equiv_{\mathcal{O}_{\leq 0}}} &= \{b, ba, bb\} \\ [\![a]\!]_{\equiv_{\mathcal{O}_{\leq 0}}} &= \{a, aa, ab\}. \end{aligned}$$

We then construct the hypothesis DFA $\mathcal{H}_{\leq 0}$, which is given in Figure 5.10a. By a partial equivalence query over $\mathcal{H}_{\leq 0}$, we confirm that $\mathcal{L}(\mathcal{H}_{\leq 0}) = \mathcal{L}_{\leq 0}(\mathcal{A})$, *i.e.*, learning the bounded behavior graph up to 0 is done. Since we cannot extract any ultimately periodic description from $\mathcal{H}_{\leq 0}$, we immediately convert it into an ROCA $\mathcal{H}_{ROCA}$ that never increases its counter. This ROCA is given in Figure 5.10b. We call $\mathbf{EQ}(\mathcal{H}_{ROCA})$. Assume that the teacher returns the word $w = aabaa$. By performing counter value queries on every prefix of the counterexample, we increase the counter limit to 2, *i.e.*, we now work with

	ε	b
ε	**no**, 0	**yes**, 0
b	**yes**, 0	**yes**, 0
a	**no**, 1	**no**, $\bot$
aa	**no**, 2	**no**, 2
aab	**no**, 2	**no**, $\bot$
$aaba$	**no**, 1	**no**, $\bot$
$aabaa$	**yes**, 0	**yes**, 0
ba	**yes**, 0	**yes**, 0
bb	**yes**, 0	**yes**, 0
ab	**no**, $\bot$	**no**, $\bot$
aaa	**no**, $\bot$	**no**, $\bot$
$aabb$	**no**, $\bot$	**no**, $\bot$
$aabab$	**no**, $\bot$	**no**, $\bot$
$aabaaa$	**yes**, 0	**yes**, 0
$aabaab$	**yes**, 0	**yes**, 0

(a) Initial table.

	ε	b	a	aa	baa
ε	**no**, 0	**yes**, 0	**no**	**no**	**yes**
b	**yes**, 0	**yes**, 0	**yes**	**yes**	**yes**
a	**no**, 1	**no**, 1	**no**	**no**	**yes**
aa	**no**, 2	**no**, 2	**no**	**no**	**yes**
aab	**no**, 2	**no**, 2	**no**	**yes**	**yes**
$aaba$	**no**, 1	**no**, 1	**yes**	**yes**	**yes**
$aabaa$	**yes**, 0	**yes**, 0	**yes**	**yes**	**yes**
ba	**yes**, 0	**yes**, 0	**yes**	**yes**	**yes**
bb	**yes**, 0	**yes**, 0	**yes**	**yes**	**yes**
ab	**no**, 1	**no**, 1	**yes**	**yes**	**yes**
aaa	**no**, $\bot$	**no**, $\bot$	**no**	**no**	**no**
$aabb$	**no**, 2	**no**, 2	**no**	**yes**	**yes**
$aabab$	**no**, 1	**no**, 1	**yes**	**yes**	**yes**
$aabaaa$	**yes**, 0	**yes**, 0	**yes**	**yes**	**yes**
$aabaab$	**yes**, 0	**yes**, 0	**yes**	**yes**	**yes**

(b) After resolving the (aa, aab, b)-$\bot$-inconsistency.

Figure 5.11: First observation tables up to two for Section 5.4.6.

$\mathcal{O}_{\leq 2}$. Furthermore, we add all the prefixes of w as new representatives.

The resulting table is given in Figure 5.11a. One can check that aa is in *Approx*(aab) but $C(aa \cdot b) = 2$ and $C(aab \cdot b) = \bot$. That is, we have a (aa, aab, b)-$\bot$-inconsistency. As we did earlier, let $u = aa, v = aab, s = b, u' = aabaa$, and $s' = \varepsilon$. We have $u \cdot s = aab$ is a prefix of $u' \cdot s' = aabaa$. Let $s'' = baa$ (observe that $u \cdot s'' = aabaa = u' \cdot s'$, as required). Since u is a proper prefix of u', we call $\mathbf{MQ}(v \cdot s'') = \mathbf{MQ}(aab \cdot baa)$, which returns **yes**, indicating that $aabbaa \in \mathcal{L}(\mathcal{A})$. As we need to check whether $aabbaa \in \mathcal{L}_{\leq 2}(\mathcal{A})$, we perform a counter value query on each prefix of $aabbaa$ and observe that the height of the word is 2. Hence, $aabbaa \in \mathcal{L}_{\leq 2}(\mathcal{A})$. We thus add all suffixes of $s'' = baa$ in $\widehat{S}$.[15] Figure 5.11b gives the obtained table. Observe that adding the new columns changed the prefix of the language encoded in the observation table and some values for C changed. For instance, $C(a \cdot b)$ is now 1, instead of $\bot$. Furthermore, the $\bot$-inconsistency is indeed resolved.

15: We highlight that we do not change S.

Since $aaba \in$ *Approx*(a) but $aaba \cdot a \notin$ *Approx*$(a \cdot a)$, we have a $(a, aaba, a)$-Σ-inconsistency. We also have that $T(a \cdot \varepsilon) \neq T(aaba \cdot \varepsilon)$. Hence, the Σ-inconsistency us resolved by adding all the suffixes of $a \cdot \varepsilon$ to both S and $\widehat{S}$. Figure 5.12 gives the resulting table, which has a (aa, aab, a)-Σ-inconsistency. As $T(aa \cdot a) \neq T(aab \cdot a)$, we add all suffixes of $a \cdot a$ to both S and $\widehat{S}$ and obtain the table of Figure 5.13.

We have a (a, aaa, ε)-$\bot$-inconsistency, which can be resolved by adding all suffixes of baa to S. Furthermore, *Approx*$(aaa) \cap R = \emptyset$, *i.e.*, we also have a aaa-openness, which is resolved by adding aaa as a new representative. Figure 5.14 gives the corresponding table.

One can check that this table is closed, Σ and $\bot$-consistent. Hence, we can

	ε	b	a	aa	baa
ε	**no**, 0	**yes**, 0	**no**, 1	**no**	**yes**
b	**yes**, 0	**yes**, 0	**yes**, 0	**yes**	**yes**
a	**no**, 1	**no**, 1	**no**, 2	**no**	**yes**
aa	**no**, 2	**no**, 2	**no**, $\perp$	**no**	**yes**
aab	**no**, 2	**no**, 2	**no**, 1	**yes**	**yes**
$aaba$	**no**, 1	**no**, 1	**yes**, 0	**yes**	**yes**
$aabaa$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**	**yes**
ba	**yes**, 0	**yes**, 0	**yes**, 0	**yes**	**yes**
bb	**yes**, 0	**yes**, 0	**yes**, 0	**yes**	**yes**
ab	**no**, 1	**no**, 1	**yes**, 0	**yes**	**yes**
aaa	**no**, $\perp$	**no**, $\perp$	**no**, $\perp$	**no**	**no**
$aabb$	**no**, 2	**no**, 2	**no**, 1	**yes**	**yes**
$aabab$	**no**, 1	**no**, 1	**yes**, 0	**yes**	**yes**
$aabaaa$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**	**yes**
$aabaab$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**	**yes**

Figure 5.12: After resolving the $(a, aaba, a)$-Σ-inconsistency of Figure 5.11b.

	ε	b	a	aa	baa
ε	**no**, 0	**yes**, 0	**no**, 1	**no**, 2	**yes**
b	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**
a	**no**, 1	**no**, 1	**no**, 2	**no**, $\perp$	**yes**
aa	**no**, 2	**no**, 2	**no**, $\perp$	**no**, $\perp$	**yes**
aab	**no**, 2	**no**, 2	**no**, 1	**yes**, 0	**yes**
$aaba$	**no**, 1	**no**, 1	**yes**, 0	**yes**, 0	**yes**
$aabaa$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**
ba	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**
bb	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**
ab	**no**, 1	**no**, 1	**yes**, 0	**yes**, 0	**yes**
aaa	**no**, $\perp$	**no**, $\perp$	**no**, $\perp$	**no**, $\perp$	**no**
$aabb$	**no**, 2	**no**, 2	**no**, 1	**yes**, 0	**yes**
$aabab$	**no**, 1	**no**, 1	**yes**, 0	**yes**, 0	**yes**
$aabaaa$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**
$aabaab$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**

Figure 5.13: After resolving the (aa, aab, a)-Σ-inconsistency of Figure 5.12.

	ε	b	a	aa	baa
ε	**no**, 0	**yes**, 0	**no**, 1	**no**, 2	**yes**, 0
b	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0
a	**no**, 1	**no**, 1	**no**, 2	**no**, $\bot$	**yes**, 0
aa	**no**, 2	**no**, 2	**no**, $\bot$	**no**, $\bot$	**yes**, 0
aab	**no**, 2	**no**, 2	**no**, 1	**yes**, 0	**yes**, 0
$aaba$	**no**, 1	**no**, 1	**yes**, 0	**yes**, 0	**yes**, 0
$aabaa$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0
aaa	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$
ba	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0
bb	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0
ab	**no**, 1	**no**, 1	**yes**, 0	**yes**, 0	**yes**, 0
$aabb$	**no**, 2	**no**, 2	**no**, 1	**yes**, 0	**yes**, 0
$aabab$	**no**, 1	**no**, 1	**yes**, 0	**yes**, 0	**yes**, 0
$aabaaa$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0
$aabaab$	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0	**yes**, 0
$aaaa$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$
$aaab$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$	**no**, $\bot$

Figure 5.14: After resolving the (a, aaa, ε)-$\bot$-inconsistency and the aaa-openness of Figure 5.13.

define the following equivalence relation $\equiv_{\mathcal{O}_{\leq 2}}$:

$$\begin{aligned}
[\![\varepsilon]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{\varepsilon\}\\
[\![b]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{b, aabaa, ba, bb, aabaaa, aabaab\}\\
[\![a]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{a\}\\
[\![aa]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{aa\}\\
[\![aab]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{aab, aabb\}\\
[\![aaba]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{aaba, ab, aabab\}\\
[\![aaa]\!]_{\equiv_{\mathcal{O}_{\leq 2}}} &= \{aaa, aaaa, aaaab\}
\end{aligned}$$

We can construct a 7-state DFA $\mathcal{H}_{\leq 2}$, which is given in Figure 5.16. By a partial equivalence query over $\mathcal{H}_{\leq 2}$, we confirm that $\mathcal{L}(\mathcal{H}_{\leq 2}) = \mathcal{L}_{\leq 2}(\mathcal{A})$, *i.e.*, learning the bounded behavior graph up to 2 is done. From that DFA, we obtain an ultimately periodic description.[16] First, we enumerate the states:

16: An algorithm to compute an ultimately periodic description is given in [NL10]. We immediately give the computed description here.

$$\begin{aligned}
\nu_0([\![\varepsilon]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = \nu_1([\![a]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = \nu_2([\![aa]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = \nu_3([\![aaa]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = 1\\
\nu_0([\![b]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = \nu_1([\![aaba]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = \nu_2([\![aab]\!]_{\equiv_{\mathcal{O}_{\leq 2}}}) = 2.
\end{aligned}$$

This allows to abstract the transitions as:

$$\begin{aligned}
\tau_0(1, a) = \tau_1(1, a) = \tau_2(1, a) &= (1, +1)\\
\tau_0(1, b) = \tau_1(1, b) = \tau_2(1, b) &= (2, 0)\\
\tau_0(2, a) = \tau_0(2, b) = \tau_1(2, b) = \tau_2(2, b) &= (2, 0)\\
\tau_1(2, a) = \tau_2(2, a) &= (2, -1).
\end{aligned}$$

Then, we construct an ROCA $\mathcal{H}$, as explained in Section 5.4.4, which is given in

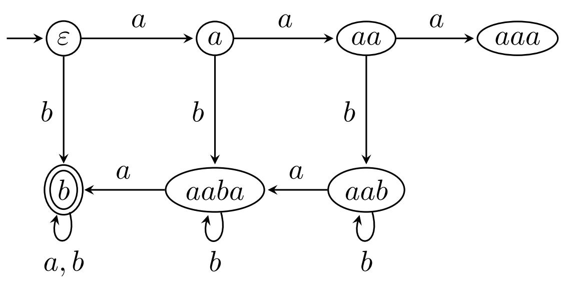


Figure 5.15: The hypothesis DFA constructed from the table of Figure 5.14.

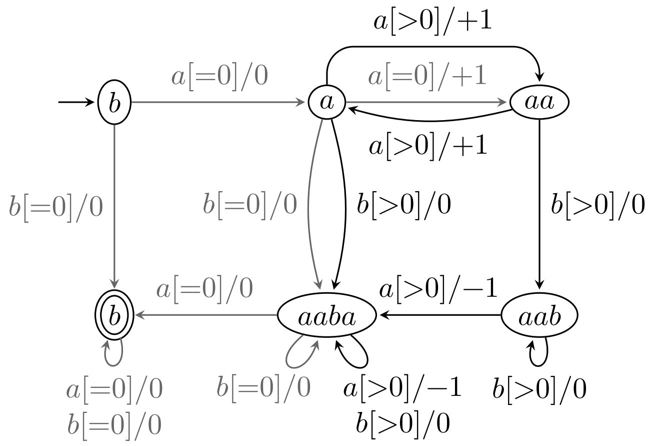


Figure 5.16: A hypothesis ROCA constructed from the DFA of Figure 5.15.

Figure 5.16. An equivalence query over $\mathcal{H}$ returns **yes**, meaning that $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{A})$, and we are done.

To conclude, we repeat our main theorem and recall that its proof can be found in Section A.8.

Theorem 5.4.1. *Let $\mathcal{A}$ be the sound ROCA of the teacher and ζ be the length of the longest counterexample returned by the teacher on (partial) equivalence queries. Then,*

- *the L^*_{ROCA} algorithm eventually terminates and returns an ROCA accepting $\mathcal{L}(\mathcal{A})$ and whose size is polynomial in $|Q^{\mathcal{A}}|$ and $|\Sigma|$,*
- *in time and space exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ, and*
- *asking a number of* **PEQ** *in $\mathcal{O}(\zeta^3)$, a number of* **EQ** *in $\mathcal{O}(|Q^{\mathcal{A}}|\zeta^2)$, and a number of* **MQ** *and* **CVQ** *exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ.*

5.5. Experimental results

We evaluated our learning algorithm L^*_{ROCA} on two types of benchmarks. The first one uses randomly generated ROCAs, while the second one focuses on a new approach to learn an ROCA that can efficiently verify whether a JSON document is valid against a given JSON schema. While there already existed several algorithms that infer a JSON schema from a collection of JSON documents (see survey [Baa+19]), this was, to the best of our knowledge, the first effort towards an approach based on automata learning techniques.

[Baa+19]: Baazizi et al. (2019), "Schemas And Types For JSON Data"

We first discuss some optimizations and implementation details, followed by the random benchmarks in Section 5.5.2. We then introduce more precisely the context of our JSON-based use case and give the results in Section 5.5.3.

5.5.1. Implementation details

We present here two optimizations regarding an observation table $\mathcal{O}_{\leq \ell} = (R, S, \hat{S}, T, C)$ up to ℓ. The first one is an efficient way to store and manipulate $\mathit{Pref}(\mathcal{O}_{\leq \ell})$, while the second one improves the computations of the approximation sets. Finally, we give our experimental framework.

Using a tree to store observations

The first optimization we consider is to store $\mathit{Pref}(\mathcal{O}_{\leq \ell})$ in a tree structure, called the *observation tree* and denoted by $\mathcal{T}_{\leq \ell}$. Then, when a new word u is added in $\mathit{Pref}(\mathcal{O}_{\leq \ell})$, it is sufficient to only update the path in $\mathcal{T}_{\leq \ell}$ leading to u. Finally, instead of directly storing the values for T and C in the table $\mathcal{O}_{\leq \ell}$, each cell of $\mathcal{O}_{\leq \ell}$ stores a pointer to a node in the tree $\mathcal{T}_{\leq \ell}$. We only give the general intuition here.

Hence, our goal is to store all the values that are needed in $\mathcal{O}_{\leq \ell}$, *i.e.*, the values for the functions T and C (on their respective domains). If we store all the information directly in $\mathcal{T}_{\leq \ell}$, having to recompute some C due to a change in $\mathit{Pref}(\mathcal{O}_{\leq \ell})$ is more immediate. Indeed, only the prefixes of a word u added to $\mathit{Pref}(\mathcal{O}_{\leq \ell})$ can potentially have new values for C. It is thus sufficient to only iterate over the ancestors of the node storing the data for u. In other words, the tree structure reduces the runtime complexity.[17]

17: The worst-case scenario does not change much, however.

When new representatives or separators have to be added in $\mathcal{O}_{\leq \ell}$, nodes are added in $\mathcal{T}_{\leq \ell}$ and the table is extended and updated with pointers to the tree. Moreover, when a word u is added in $\mathcal{T}_{\leq \ell}$, all the prefixes x of u must also be added, even if $x \notin (R \cup R\Sigma)\hat{S}$. As a consequence, the number of nodes in $\mathcal{T}_{\leq \ell}$ will be greater than $\left|(R \cup R\Sigma)\hat{S}\right|$ since all the prefixes are stored in the tree. In other words, the tree increases the memory consumption of the algorithm but significantly improves the run time.

Efficient computation of approximation sets

Let us now discuss how to efficiently compute the approximation sets. Let $u, v \in R \cup R\Sigma$. If there exists $s \in S$ such that $T(us) \neq T(vs)$, it immediately follows that $v \notin \mathit{Approx}(u)$. Thus, when computing $\mathit{Approx}(u)$, we know that it is not interesting to consider words v that do not agree with u on the values of T. To do so, we define a relation that groups together words having the same row contents, when we only consider T. For two words $u, v \in R \cup R\Sigma$,

Definition 5.4.9. Let $\mathcal{O}_{\leq \ell}$ be an observation table up to ℓ, and $u, v \in R \cup R\Sigma$. Then, $u \in \mathit{Approx}(v)$ if and only if:

- for all $s \in S$, $T(us) = T(vs)$, and
- for all $s \in S$, if $C(us) \neq \perp$ and $C(vs) \neq \perp$, then $C(us) = C(vs)$.

The set $\mathit{Approx}(v)$ is called an *approximation set*.

we write $u \sim_{\mathcal{O}_{\leq \ell}} v$ if and only if for all $s \in S$, $T(us)$ and $T(vs)$ are equal. The approximation sets can thus be defined as:

Definition 5.5.1 (Optimization of Definition 5.4.9)**.** Let $\mathcal{O}_{\leq \ell}$ be an observation table up to ℓ, $u \in R \cup R\Sigma$, and $v \in [\![u]\!]_{\sim_{\mathcal{O}_{\leq \ell}}}$. Then, $v \in \mathit{Approx}(u)$ if

$$\forall s \in S : C(us) \neq \bot \wedge C(vs) \neq \bot \Rightarrow C(us) = C(vs).$$

With this definition, to compute $\mathit{Approx}(u)$, it is enough to iterate over the elements $v \in [\![u]\!]_{\sim_{\mathcal{O}_{\leq \ell}}}$ (instead of all $v \in R \cup R\Sigma$).

One can efficiently store each equivalence class of $\sim_{\mathcal{O}_{\leq \ell}}$ using a set, in order to obtain a polynomial time complexity for lookup, addition, and removal of an element, in the size of the table.[18] Moreover, we can also store the *Approx* sets instead of recomputing them from scratch each time.

18: The complexity can be reduced to a constant time, in average, if we assume the set relies on hash tables.

Notice that updating the classes of $\sim_{\mathcal{O}_{\leq \ell}}$ can be easily done. Suppose a value $T(us)$ (with $u \in R \cup R\Sigma$ and $s \in S$) is modified, and let $[\![v]\!]_{\sim_{\mathcal{O}_{\leq \ell}}}$ be the class of u before the modification. We have to remove u from the set for $[\![v]\!]_{\sim_{\mathcal{O}_{\leq \ell}}}$ and then add it to the set for $[\![u]\!]_{\sim_{\mathcal{O}_{\leq \ell}}}$. This can be done in polynomial time.

Finally, we need to compute the *Approx* sets only when the table has to be made closed, Σ- and $\bot$-consistent (since new elements are added in $R \cup R\Sigma$, S, or $\widehat{S}$). Notice that not all approximation sets are modified and thus should not be recomputed. This happens for $\mathit{Approx}(u)$ when $[\![u]\!]_{\sim_{\mathcal{O}_{\leq \ell}}}$ is not modified and the values $C(us)$ remain unchanged.

Experimental framework

The ROCAs and the learning algorithm were implemented by extending the well-known Java libraries AutomataLib [19] and LearnLib [20] [IHS15]. These modifications and the benchmarks can be consulted on our GitHub repositories.[21] The server used for the computations ran Debian 10 over Linux 5.4.73-1-pve with a 4-core Intel® Xeon® Silver 4214R Processor with 16.5M cache, and 64GB of RAM. Moreover, we used OpenJDK version 11.0.12.

19: https://learnlib.de/pages/automatalib

20: https://learnlib.de/

[IHS15]: Isberner et al. (2015), "The Open-Source LearnLib - A Framework for Active Automata Learning"

21: https://github.com/DocSkellington/automatalib/, https://github.com/DocSkellington/learnlib/, and https://github.com/DocSkellington/LStar-ROCA-Benchmarks. A Zenodo artifact is available at https://zenodo.org/records/5569340.

5.5.2. Random realtime one-counter automata

We discuss our benchmarks based on randomly generated ROCAs. We begin by explaining how this random generation works, followed by how we check equivalence of two ROCAs. We finally comment our results.

Random generation of ROCAs

An ROCA $\mathcal{A}$ with given size $n = |Q^{\mathcal{A}}|$ is randomly generated as follows:[22]

- for all $q \in Q^{\mathcal{A}}$, q has a probability 0.5 of being final,

22: All random draws are assumed to come from a uniform distribution.

Algorithm 5.2: Algorithm for checking the equivalence of two ROCAs.

Require: Two ROCAs $\mathcal{A}$ and $\mathcal{A}_\alpha$, and the period k of the description α used to construct $\mathcal{A}_\alpha$
Ensure: Returns **yes** if our approximation for $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_\alpha)$ holds, or a counterexample

```
1:  $Q \leftarrow$ a queue initialized with $(\varepsilon, (q_0^{\mathcal{A}}, 0), (q_0^{\mathcal{A}_\alpha}, 0))$
2:  $\ell \leftarrow (|Q^{\mathcal{A}}| \cdot |Q^{\mathcal{A}_\alpha}|)^2$
3:  repeat
4:      while $Q$ is not empty do
5:          $(w, c_{\mathcal{A}}, c_{\mathcal{A}_\alpha}) \leftarrow$ the next element in $Q$        ▷ We pop the tuple from $Q$
6:          for all $a \in \Sigma$ do
7:              Let $(q, n) \in Q^{\mathcal{A}} \times \mathbb{N}$ such that $c_{\mathcal{A}} \xrightarrow{a} (q, n) \in \mathit{cruns}(\mathcal{A})$
8:              Let $(p, m) \in Q^{\mathcal{A}_\alpha} \times \mathbb{N}$ such that $c_{\mathcal{A}_\alpha} \xrightarrow{a} (p, m) \in \mathit{cruns}(\mathcal{A}_\alpha)$
9:              if $(q \in F^{\mathcal{A}} \wedge n = 0) \Leftrightarrow \neg(p \in F^{\mathcal{A}_\alpha} \wedge m = 0)$ then
10:                 return $wa$        ▷ $wa$ is a witness that $\mathcal{A}$ and $\mathcal{A}_\alpha$ are not equivalent
11:             else if $n \leq \ell \wedge m \leq \ell$ and $(wa, (q, n), (p, m))$ has not yet been seen then
12:                 Add $(wa, (q, n), (p, m))$ in $Q$
13:     $\ell \leftarrow \ell + k$
14: until we stop with probability $0.5$
15: return yes
```

- for all $q \in Q^{\mathcal{A}}$ and $a \in \Sigma$, we have $q \xrightarrow[c]{a[>0]} p$ with p a random state in $Q^{\mathcal{A}}$ and c a random counter operation in $\{-1, 0, +1\}$. Moreover, we define $q \xrightarrow[c]{a[=0]} p$ in a similar way except that $c \in \{0, +1\}$.

Since this generation does not guarantee the produced ROCA has n reachable states, we produce 100 ROCAs and select the ROCA with a number of reachable states that is maximal. However, note that it is still possible the resulting ROCA does not have n (co)-reachable states.

Equivalence of two ROCAs

The language equivalence problem of ROCAs is known to be decidable and NL-complete [BGJ14]. Unfortunately, the algorithm described in [BGJ14] is difficult to implement.[23] Instead, we use an "approximate" equivalence oracle for our experiments.[24]

[BGJ14]: Böhm et al. (2014), "Bisimulation equivalence and regularity for real-time one-counter automata"

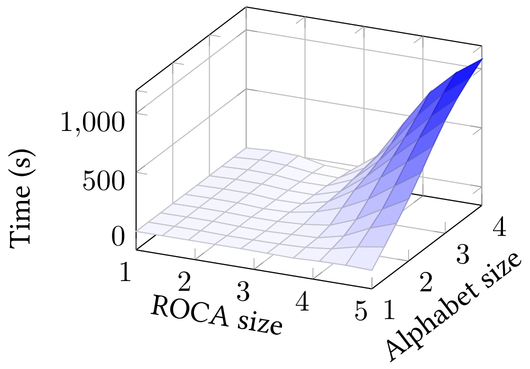
23: In short, Böhm *et al.* show that there exist some polynomials satisfying some properties and that allow to decide the equivalence. However, there is no algorithm provided to compute these polynomials.

24: The teacher might, with some small probability, answer with false positives but never with false negatives.

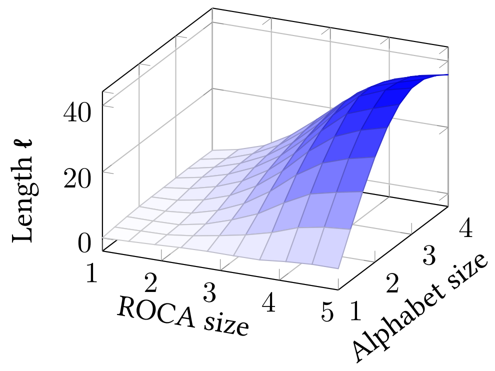
Let $\mathcal{A}$ and $\mathcal{B}$ be two ROCAs such that $\mathcal{B}$ is the learned ROCA from a periodic description with period k. The algorithm explores the configuration space of both ROCAs in parallel. If, at some point, it reaches a pair of configurations such that one is accepting and the other not, then we have a counterexample. However, to have an algorithm that eventually stops, we need to bound the counter value of the configurations to explore. Our approach is to first explore up to counter value $|\mathcal{A} \times \mathcal{B}|^2$ (in view of [BGJ14, Proposition 18] about shortest accepting runs in an ROCA). If no counterexample is found, we add k to the bound and, with probability 0.5, a new exploration is done up to the new bound. We repeat this whole process until we find a counterexample or until the random draw forces us to stop. This is summarized in Algorithm 5.2.

$\lvert Q^{\mathcal{A}}\rvert$	$\lvert\Sigma\rvert$	TO (20 min)
4	1	0
4	2	5
4	3	16
4	4	41
5	1	0
5	2	23
5	3	55
5	4	83

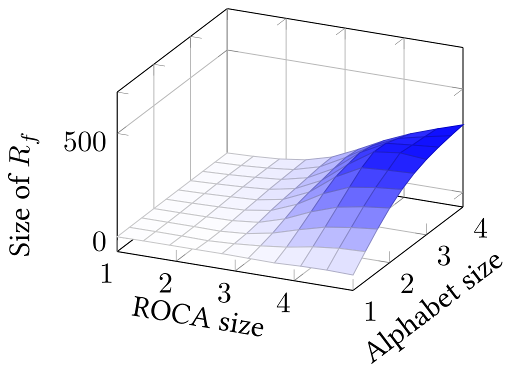
Table 5.1: Number (over 100) of executions with a timeout (TO). The executions for the missing pairs $(\lvert Q^{\mathcal{A}}\rvert, \lvert\Sigma\rvert)$ could all finish.



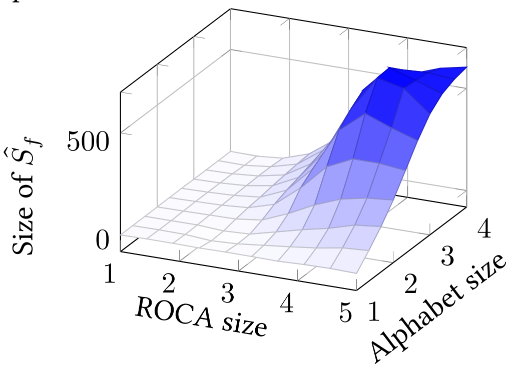
(a) Mean of the total time taken by L^*_{ROCA}.



(b) Mean of the length ζ of the longest counterexample.



(c) Mean of the final size of R.



(d) Mean of the final size of $\hat{S}$.

Figure 5.17: Results for the benchmarks based on random ROCAs.

Results

For our random benchmarks, we let the size $\lvert Q^{\mathcal{A}}\rvert$ of the ROCA $\mathcal{A}$ vary between one and five, and the size $\lvert\Sigma\rvert$ of the alphabet between one and four. For each pair $(\lvert Q^{\mathcal{A}}\rvert, \lvert\Sigma\rvert)$, we execute the learning algorithm on 100 ROCAs (generated as explained above). We set a timeout of 20 minutes and a memory limit of 16GB. The number of executions with a timeout is given in Table 5.1 (we do not give the pairs $(\lvert Q^{\mathcal{A}}\rvert, \lvert\Sigma\rvert)$ where every execution could finish).

The mean of the total time taken by the algorithm is given in Figure 5.17a.

One can see that it has an exponential growth in both sizes $|Q^{\mathcal{A}}|$ and $|\Sigma|$. Note that executions with a timeout had their execution time set to 20 minutes, in order to highlight the curve. Let us now drop all the executions with a timeout. The mean length of the longest counterexample provided by the teacher for (partial) equivalence queries is presented in Figure 5.17b and the final size of the sets R and $\widehat{S}$ are presented in Figures 5.17c and 5.17d. Note that the curves go down due to the limited number of remaining executions (for instance, the ones that could finish did not require long counterexamples). We can see that $\widehat{S}$ grows larger than R. We conclude that these empirical results confirm the theoretical complexity claims from Theorem 5.4.1.[25]

25: This also confirms the growth of the number of representatives and separators stated in Section A.8.

We conclude this section on the benchmarks based on random ROCAs by highlighting the fact that our random ROCAs do not reflect real-life automata. That is, in more concrete cases, ROCAs may have a natural structure that is induced by the problem. Thus, results may be very different. Hence, we applied our learning algorithm to a more realistic use case.

5.5.3. JSON documents and JSON Schemas

Let us now discuss the second set of benchmarks, which is a proof of concept for our learning algorithm based on JSON documents [Bra17]. This format is the currently most popular one used for exchanging information on the web. We give here the main ideas. A more thorough introduction is given in Part III, alongside a validation algorithm that can handle more cases.

[Bra17]: Bray (2017), “The JavaScript Object Notation (JSON) Data Interchange Format”

Our goal is to construct an ROCA that can validate a JSON document, according to some given constraints. This use case is inspired by [CR04] but applied on a more recent format.

[CR04]: Chitic et al. (2004), “On Validation of XML Streams Using Finite State Machines”

A *JSON document* is a text document that follows a specific structure. Namely, five different types of data can be present in a document:[26]

26: The standard specifies a sixth type (*null*) that we do not consider in our use case.

- An *object* is an unordered collection of pairs key-value where key is a finite string and value can be any of the five different data types. An object must start with `{` and end with `}`.
- An *array* is an ordered collection of values. Again, a value can be any of the five different data types. An array must start with `[` and end with `]`.
- A *string* is a finite sequence of any Unicode characters and must start and end with `"`.
- A *number* is any positive or negative decimal real number. In particular, an *integer* is any positive or negative number without a decimal part.
- A *boolean* can be `true` or `false`.

A JSON document must start with an object.

In the context we consider, the constraints a document must satisfy are given by a *JSON Schema* which is itself a JSON document describing the kind of values that must be associated to each key. An example of a schema is given in Figure 5.18. JSON schemas are described more thoroughly on the official website[27] and in Part III. They can be seen as a counterpart to DTDs for XML documents. A schema can allow a recursive structure (to model a tree, for instance). The ranges for the different values can be restricted. For example,

27: `https://json-schema.org`

```json
{
  "title": "Fast",
  "type": "object",
  "required": ["string", "double", "integer", "boolean", "object", "array"],
  "additionalProperties": false,
  "properties": {
    "string": {"type": "string"},
    "double": {"type": "number"},
    "integer": {"type": "integer"},
    "boolean": {"type": "boolean"},
    "object": {
      "type": "object",
      "required": ["anything"],
      "additionalProperties": false,
      "properties": {
        "anything": {
          "type": ["number", "integer", "boolean", "string"]
        }
      }
    },
    "array": {
      "type": "array",
      "items": {"type": "string"},
      "minItems": 2
    }
  }
}
```

Figure 5.18: Example of a JSON schema.

one can force an integer to be in the interval $[0, 10]$, or to be a multiple of two, and so on.

Learning a JSON Schema as an ROCA

We propose to validate a JSON document against a given JSON schema as follows. The learner learns an ROCA from the schema. With this ROCA, one can easily decide whether a JSON document is valid against the schema.

In this learning process, we suppose the teacher knows the target schema and the queries are specialized as follows:

Membership query The learner provides a JSON document and the teacher returns true if and only if the document is valid for the schema.

Counter value query The learner provides a JSON document and the teacher returns the number of unmatched `{` and `[`. Adding the two values is a heuristic abstraction that allows us to summarize two-counter information into a single counter value. Importantly, the abstraction is a design choice regarding our implementation of a teacher for these experiments and not an assumption made by our learning algorithm. The approach given in Part III lifts this constraint.

Partial equivalence query The learner gives a DFA and a counter limit ℓ. The teacher randomly generates an a-priori fixed number of documents with a height not exceeding the counter limit ℓ and checks whether the DFA and the schema both agree on the documents' validity. If a disagreement is noticed, the incorrectly classified document is returned.

Equivalence query The learner provides an ROCA. The teacher also generates an a-priori number of randomly generated documents (but, this time, without a bound on the height) and verifies that the ROCA and the schema both agree on the documents' validity. If one is found, an incorrectly classified document is returned.

Note that the randomness of the (partial) equivalence queries implies that the learned ROCA may not exactly recognize the set of documents accepted by the schema. Setting the number of generated documents to be a large number helps reducing the probability that an incomplete ROCA is learned. One could also control the randomness of the generation to force some important keys to appear, even if these keys are not required in the schema.

In order for an ROCA to be learned in a reasonable time, some abstractions must be made mainly for reducing the alphabet size.

- If an object has a key named `key`, we consider the sequence of characters `"key"` as a single alphabet symbol.
- Strings, integers, and numbers are abstracted as follows. All strings are replaced by `"\S"`, all integers by `"\I"`, and all numbers by `"\D"`. Booleans are left as-is since they can only take two values (`true` or `false`).
- The symbols `,`, `{`, `}`, `[`, `]`, `:` are all considered as different alphabet symbols (note that since `"` is considered directly into the keys' symbols or the values' symbols, that symbol does not appear here).

Moreover, notice that the alphabet is not known at the start of the learning process (due to the fact that keys can be any strings). Therefore we slightly modify the learning algorithm to support growing alphabets. More precisely, the learner's alphabet starts with the symbols { and } (to guarantee we can at least produce a syntactically valid JSON document for the first partial equivalence query) and is augmented each time a new symbol is seen.

A last abstraction was applied for our benchmarks: we assume that each object is composed of an ordered (instead of unordered) collection of pairs key-value. It is to be denoted by that the learning algorithm can learn without this restriction. However it requires substantially more time as all possible orders inside each object must be considered (and they all induce a different set of states in the ROCA). Part III will present a validation algorithm (based on automata learning) where objects remain unordered. The use case we show here only serves as a benchmark for ROCA learning and should not be considered as a usable validation algorithm for JSON documents.

Results

We considered three JSON schemas. The first one is the document from Figure 5.18 which lists all possible types of values (*i. e.*, it contains an integer, a double, and so on). The second one is a real-world JSON schema[28] used by a code coverage tool called Codecov.[29] Finally, the third schema encodes a recursive list, *i. e.*, an object containing a list with at most one object defined

28: The schema was downloaded from JSON Schema Store `https://www.schemastore.org/json/`. We modified the file to remove all constraints of type "enum".

29: `https://about.codecov.io`

Schema	TO (1h)	Time (s)	ζ	$\lvert R \rvert$	$\lvert \hat{S} \rvert$	$\lvert Q^{\mathcal{A}} \rvert$	$\lvert \Sigma \rvert$
Figure 5.18	0	16.39	31.00	55.55	32.00	33.00	19.00
Codecov	27	1045.64	12.99	57.84	33.74	44.29	14.70
Figure 5.19	19	922.19	49.49	171.94	50.49	51.16	9.00

Table 5.2: Results for JSON benchmarks.

```
{
    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "children": {
            "type": "array",
            "maxItems": 1,
            "items": {"$ref": "#"}
        }
    },
    "required": ["name"],
    "additionalProperties": false
}
```

Figure 5.19: Recursive JSON schema.

recursively. This last example is used to force the behavior graph to be ultimately periodic, and is given in Figure 5.19. The `{"$ref": "#"}` indicates that the each item in the array `children` is defined as the top object, *i.e.*, we have a recursive definition.

The Table 5.2 gives the results of the benchmarks, obtained by fixing the number of random documents by (partial) equivalence query to be 1000. For each schema, 100 experiments were conducted with a time limit of one hour by execution. We can see that real-world JSON schemas and recursively-defined schemas can be both learned by our approach. One last interesting statistics we can extract from the results is that the number of representatives is larger than the number of separators, unlike for the random benchmarks.

5.6. Conclusion

We have presented a new learning algorithm $L^*_{\text{ROCA}}$for realtime one-counter automata, using membership, counter value, partial equivalence, and equivalence queries. The algorithm executes in exponential time and space, and requires at most an exponential number of queries. We have implemented this algorithm and evaluated it on two benchmarks.

As future work, we believe one might be able to remove the use of partial equivalence queries. In this direction, perhaps replacing our use of Neider and Löding's VOCA algorithm by Isberner's TTT algorithm [IHS14b; Isb15] might help, due to the differences in how the gathered knowledge is stored in both algorithms. The $L^{\#}$ algorithm of Vaandrager *et al.* [Vaa+22], as it does not infer an equivalence relation but instead focuses on identifying the

[IHS14b]: Isberner et al. (2014), "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning"

[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness" differences in behavior of two states (see Section 3.3) may also simplify the learning algorithm for ROCAs.

Another interesting direction concerns lowering the (query) complexity of our algorithm. In [RS93], it is proved that L^* algorithm [Ang87] can be modified so that adding a single separator after a failed equivalence query is enough to update the observation table. This would remove the suffix-closedness requirements on the separator sets S and $\hat{S}$. It is not immediately clear to us whether the definition of $\bot$-consistency presented here holds in that context. Further optimizations, such as discrimination tree-based algorithms (*e.g.*, Kearns and Vazirani's algorithm [KV94]), also do not need the separator set to be suffix-closed.

[RS93]: Rivest et al. (1993), "Inference of Finite Automata Using Homing Sequences"

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

[KV94]: Kearns et al. (1994), *An Introduction to Computational Learning Theory*

It would also be interesting to directly learn the one-counter language instead of an ROCA. Indeed, our algorithm learns some ROCA that accepts the target language. It would be desirable to learn some canonical representation of the language (*e.g.*, a minimal automaton, for some notion of minimality).

Finally, as far as we know, there currently is no active learning algorithm for deterministic one-counter automata (where ε-transitions are allowed).

Technical Details and Proofs of Chapter 5 A.

This chapter, based on [BPS22], contains the technical details and proofs that were not given in Chapter 5. That is, it serves as the appendix of the previous chapter.

[BPS22]: Bruyère et al. (2022), "Learning Realtime One-Counter Automata"

Chapter contents

A.1. Proof of Theorem 5.3.10

Theorem 5.3.10. *Let $\mathcal{A}$ be a sound ROCA, $BG(\mathcal{A})$ be its behavior graph, $\widetilde{\mathcal{A}}$ be the corresponding VOCA accepting $\widetilde{L}$, and $BG(\widetilde{L})$ be the behavior graph of $\widetilde{L}$. Then,*

- *$BG(\mathcal{A})$ and $BG(\widetilde{L})$ are isomorphic up to $\lambda_{\mathcal{A}}$ and $\tilde{\lambda}$, and*
- *the isomorphism respects the counter values (i. e., level membership) and both offset and period of periodic descriptions.*

Let $L = \mathcal{L}(\mathcal{A})$. In order to show that the two behavior graphs are isomorphic, it is sufficient to prove that $\sim_{\mathcal{A}}$ and $\sim_{\widetilde{L}}$ agree, *i. e.*,

$$\forall u, v \in \mathit{Pref}(L) : u \sim_{\mathcal{A}} v \Leftrightarrow \lambda_{\mathcal{A}}(u) \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v).$$

Indeed, as, by Lemma 5.3.9,

$$\mathit{Pref}(\widetilde{L}) = \mathit{Pref}(\mathcal{L}(\widetilde{\mathcal{A}})) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathit{Pref}(L)\},$$

we will get the required one-to-one correspondence between the states $[\![u]\!]_{\sim_{\mathcal{A}}}$ of $BG(\mathcal{A})$ and the states $[\![\lambda_{\mathcal{A}}(u)]\!]_{\sim_{\widetilde{L}}}$ of $BG(\widetilde{L})$. Notice that this correspondence respects the counter values of the states since all words in state $[\![u]\!]_{\sim_{\mathcal{A}}}$ have the same counter value $cv^{\mathcal{A}}(u)$ which is equal (by Lemma 5.3.9) to the counter value $cv(\lambda_{\mathcal{A}}(u))$ of all words in state $[\![\lambda_{\mathcal{A}}(u)]\!]_{\sim_{\widetilde{L}}}$. In other words, both $[\![\lambda_{\mathcal{A}}(u)]\!]_{\sim_{\widetilde{L}}}$ and

Lemma 5.3.9. Let $\mathcal{A}$ be a sound ROCA and $\widetilde{\mathcal{A}}$ be its corresponding VOCA. We have:

- $\forall u \in \Sigma^* : cv^{\mathcal{A}}(u) = cv(\lambda_{\mathcal{A}}(u))$,
- $\mathcal{L}(\widetilde{\mathcal{A}}) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathcal{L}(\mathcal{A})\}$, and
- $\mathit{Pref}(\mathcal{L}(\widetilde{\mathcal{A}})) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathit{Pref}(\mathcal{L}(\mathcal{A}))\}$.

$[\![u]\!]_{\sim_{\mathcal{A}}}$ are in level $cv^{\mathcal{A}}(u) = cv(\lambda_{\mathcal{A}}(u))$ of their corresponding behavior graphs. That is, the second part of the theorem follows directly from the first.

$\sim_{\mathcal{A}}$ implies $\sim_{\widetilde{L}}$. Let $u, v \in \mathit{Pref}(L)$ be such that $u \sim_{\mathcal{A}} v$, *i. e.*, for all $w \in \Sigma^*$,

$$u \cdot w \in L \Leftrightarrow v \cdot w \in L$$

and

$$u \cdot w, v \cdot w \in \mathit{Pref}(L) \Rightarrow cv^{\mathcal{A}}(u \cdot w) = cv^{\mathcal{A}}(v \cdot w).$$

We show that $\lambda_{\mathcal{A}}(u) \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v)$, *i. e.*, for all $\widetilde{w} \in \widetilde{\Sigma}^*$, it holds that $\lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \in \widetilde{L}$ if and only if $\lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \widetilde{L}$. Let $\widetilde{w} \in \widetilde{\Sigma}^*$ and assume $\lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \in \widetilde{L}$. By definition of $\widetilde{\mathcal{A}}$, there exists $x \in \Sigma^*$ such that $u \cdot x \in L$ and $\lambda_{\mathcal{A}}(u \cdot x) = \lambda_{\mathcal{A}}(u) \cdot \widetilde{w}$. Since $u \sim_{\mathcal{A}} v$, we get $v \cdot x \in L$ and for all $y \in \mathit{Pref}(x)$, $cv^{\mathcal{A}}(u \cdot y) = cv^{\mathcal{A}}(v \cdot y)$. By definition of $\widetilde{\mathcal{A}}$, it follows that $\lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \widetilde{L}$, meaning that

$$\lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \in \widetilde{L} \Rightarrow \lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \widetilde{L}.$$

Using a similar argument, we prove that

$$\lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \widetilde{L} \Rightarrow \lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \in \widetilde{L},$$

meaning that $\lambda_{\mathcal{A}}(u) \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v)$.

$\sim_{\widetilde{L}}$ implies $\sim_{\mathcal{A}}$. We now show that

$$\forall \lambda_{\mathcal{A}}(u), \lambda_{\mathcal{A}}(v) \in \mathit{Pref}(\widetilde{L}) : \lambda_{\mathcal{A}}(u) \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v) \Rightarrow u \sim_{\mathcal{A}} v.$$

First, let us show that for all $w \in \Sigma^*$, we have $u \cdot w \in L$ if and only if $v \cdot w \in L$. Assume that $u \cdot w \in L$. Then, $\lambda_{\mathcal{A}}(u \cdot w) = \lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \in \widetilde{L}$ with $\widetilde{w} \in \widetilde{\Sigma}^*$. As $\lambda_{\mathcal{A}}(u) \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v)$, we get $\lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \widetilde{L}$. Hence by discarding with $\widetilde{\lambda}$ the index in $\{c, r, \mathit{int}\}$ of each symbol of $\lambda_{\mathcal{A}}(v) \cdot \widetilde{w}$, we obtain that $v \cdot w \in L$. The other implication is proved similarly and we have thus proved that $u \cdot w \in L \Leftrightarrow v \cdot w \in L$.

Second, let us show that

$$\forall w \in \Sigma^* : u \cdot w, v \cdot w \in \mathit{Pref}(L) \Rightarrow cv^{\mathcal{A}}(u \cdot w) = cv^{\mathcal{A}}(v \cdot w).$$

Let $w \in \Sigma^*$ and $\widetilde{w} \in \widetilde{\Sigma}^*$ be such that $\lambda_{\mathcal{A}}(u \cdot w) = \lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \in \mathit{Pref}(\widetilde{L})$. It follows that $\lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v) \cdot \widetilde{w}$ and $\lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \mathit{Pref}(\widetilde{L})$ since $\lambda_{\mathcal{A}}(u) \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v)$. From

$$\lambda_{\mathcal{A}}(u) \cdot \widetilde{w} \sim_{\widetilde{L}} \lambda_{\mathcal{A}}(v) \cdot \widetilde{w}$$

and

$$\lambda_{\mathcal{A}}(v) \cdot \widetilde{w}, \lambda_{\mathcal{A}}(v) \cdot \widetilde{w} \in \mathit{Pref}(\widetilde{L}),$$

we deduce $cv(\lambda_{\mathcal{A}}(u) \cdot \widetilde{w}) = cv(\lambda_{\mathcal{A}}(v) \cdot \widetilde{w})$ by Lemma 4.3.13. We also have that $\lambda_{\mathcal{A}}(v) \cdot \widetilde{w} = \lambda_{\mathcal{A}}(v \cdot w)$. Hence, by Lemma 5.3.9, we get $cv^{\mathcal{A}}(uw) = cv^{\mathcal{A}}(vw)$.

Lemma 4.3.13. For a VOCL L and $u, v \in \mathit{Pref}(L)$ such that $u \sim_L v$, $cv(u) = cv(v)$.

Lemma 5.3.9. Let $\mathcal{A}$ be a sound ROCA and $\widetilde{\mathcal{A}}$ be its corresponding VOCA. We have:

- $\forall u \in \Sigma^* : cv^{\mathcal{A}}(u) = cv(\lambda_{\mathcal{A}}(u))$,
- $\mathcal{L}(\widetilde{\mathcal{A}}) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathcal{L}(\mathcal{A})\}$, and
- $\mathit{Pref}(\mathcal{L}(\widetilde{\mathcal{A}})) = \{\lambda_{\mathcal{A}}(u) \mid u \in \mathit{Pref}(\mathcal{L}(\mathcal{A}))\}$.

Conclusion. We established the one-to-one correspondence between the states of $BG(\mathcal{A})$ and the states of $BG(\widetilde{L})$. Notice that it puts in correspondence the initial (resp. final) states of both behavior graphs. We have also a correspondence with respect to the transitions (up to $\lambda_{\mathcal{A}}$ and $\widetilde{\lambda}$). Indeed, this follows from how both transition functions $\delta^{BG(\mathcal{A})}$ and $\delta^{BG(\widetilde{L})}$ are defined.

A.2. Proof of Proposition 5.3.12

Proposition 5.3.12. *Let $BG(\mathcal{A})$ be the behavior graph of some sound ROCA $\mathcal{A}$ and $\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega$ be an ultimately periodic description of $BG(\mathcal{A})$ with offset m and period k. Then, one can construct an ROCA $\mathcal{A}_\alpha$ from α such that*

- *$\mathcal{L}(\mathcal{A}_\alpha) = \mathcal{L}(\mathcal{A})$, and*
- *the size of $\mathcal{A}_\alpha$ is polynomial in m, k and $width(BG(\mathcal{A}))$.*

To prove this proposition, we first explicitly give the construction of the ROCA. Let $L = \mathcal{L}(\mathcal{A}) = \mathcal{L}(BG(\mathcal{A}))$, K be the width of $BG(\mathcal{A})$, and $\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega$ be an ultimately periodic description of $BG(\mathcal{A})$. Recall that the mappings τ_ℓ used in α are defined as

$$\tau_\ell : \{1, \dots, K\} \times \Sigma \rightharpoonup \{1, \dots, K\} \times \{-1, 0, +1\}.$$

Let us explain how to construct an ROCA $\mathcal{A}_\alpha$ accepting L. In short, as long as we remain in the levels 0 to $m-1$, the counter value is never modified. That is, only the periodic part of α induces increments or decrements of the counter. To ease the writing of the proof, we assume that all transitions are defined, *i. e.*, $\delta^{\mathcal{A}}$ is total. Moreover, we assume that $\mathcal{A}$ has a single bin state, denoted by $\bot$.[1] We have two cases, depending on whether k is null.

1: If $\delta^{\mathcal{A}}$ is not total, it is sufficient to define the missing transitions as $q \xrightarrow[0]{a[g]} \bot$.

A.2.1. When the width is null

First, assume $k = 0$, *i. e.*, $\alpha = \tau_0 \dots \tau_{m-1}$. Then, $BG(\mathcal{A})$ is finite[2], meaning that L is actually a regular language. We thus construct $\mathcal{A}_\alpha$ as a DFA (*i. e.*, an ROCA that does not modify its counter). In other words, all counter operations are 0, meaning that we only have to define the transitions using the guard $=0$. Let us thus construct the ROCA $\mathcal{A}_\alpha$ such that

2: As $m-1$ is the highest level, *i. e.*, the counter value is bounded

- the states stores the current level and the number of the current equivalence class within that level (given by the enumeration ν_ℓ), *i. e.*,

$$Q^{\mathcal{A}_\alpha} = \{0, \dots, m-1\} \times \{1, \dots, K\},$$

- we know that $cv^{\mathcal{A}}(\varepsilon) = 0$ (by definition of the semantics of $\mathcal{A}$), so

$$q_0^{\mathcal{A}_\alpha} = (0, \nu_0([\![\varepsilon]\!]_{\sim_{\mathcal{A}}})),$$

- likewise, every word $w \in L$ is such that $cv^{\mathcal{A}}(w) = 0$, so
$$F^{\mathcal{A}_\alpha} = \{(0, [\![w]\!]_{\sim_{\mathcal{A}}}) \mid w \in L\},$$
- the transition function is defined naturally over those states, since we never modify the counter: for all $q \in \{1, \dots, K\}$, $\ell \in \{0, \dots, m-1\}$, and $a \in \Sigma$, let $(p, c) = \tau_\ell(q, a)$ and define
$$(\ell, q) \xrightarrow[c]{a[=0]} (\ell + c, p).$$

It is not hard to see that $\mathcal{A}_\alpha$ is sound and accepts L. Moreover, $\left|Q^{\mathcal{A}_\alpha}\right|$ is bounded by $m \cdot K$, *i. e.*, the size of the ROCA is polynomial in m and $width(BG(\mathcal{A}))$. That is, Proposition 5.3.12 is easily obtained when $k = 0$.

A.2.2. When the width is not null

Now, assume $k \neq 0$, *i. e.*, α has a periodic part. As said above, we construct $\mathcal{A}_\alpha$ such that the counter value remains zero as long as we remain within the initial part of the description α (*i. e.*, in the levels 0 to $m - 1$). In the periodic part, the counter operations follow the changes in the levels. The idea is similar to the previous case, except that we use a modulo-k ($+m$) counter in the states for the periodic part. That is, $\mathcal{A}_\alpha$ is constructed such that

- the states stores the current level (modulo k) and the number of the current equivalence class within that level (given by the enumeration ν_ℓ), *i. e.*,
$$Q^{\mathcal{A}_\alpha} = \{0, \dots, m + k - 1\} \times \{1, \dots, K\} \uplus \{\bot\},$$
- the level of the initial state is necessarily zero:
$$q_0^{\mathcal{A}_\alpha} = (0, \nu_0([\![\varepsilon]\!]_{\sim_{\mathcal{A}}})),$$
- the final states are all in the level zero:
$$F^{\mathcal{A}_\alpha} = \{(0, [\![w]\!]_{\sim_{\mathcal{A}}}) \mid w \in L\},$$
- the transition function requires more care, this time, as we must update the counter in the periodic part of α. Let $q \in \{1, \dots, K\}$ and $a \in \Sigma$.
  - First, the transitions of the initial part: for all $\ell \in \{0, \dots, m-1\}$, let $(p, c) = \tau_\ell(q, a)$ and define
  $$(\ell, q) \xrightarrow[c]{a[=0]} (\ell + c, p).$$
  - When we are at the level m (*i. e.*, the boundary between the initial and the periodic parts), we must check whether we need to go the initial part (*i. e.*, the level $m - 1$) or loop back to the end of the periodic part (*i. e.*, the level $m + k - 1$). It is sufficient to check whether the counter value is zero, due to the rest of this construction. Let

$(p, c) = \tau_m(q, a)$ and define the transitions as follows:

$$\begin{aligned} c = 0 \Rightarrow (m, q) &\xrightarrow[0]{a[=0]} (m, p) \\ \wedge (m, q) &\xrightarrow[0]{a[>0]} (m, p) \\ c = +1 \Rightarrow (m, q) &\xrightarrow[+1]{a[=0]} (m + (1 \bmod k), p) \\ \wedge (m, q) &\xrightarrow[+1]{a[>0]} (m + (1 \bmod k), p) \\ c = -1 \Rightarrow (m, q) &\xrightarrow[-1]{a[=0]} (m - 1, p) \\ \wedge (m, q) &\xrightarrow[-1]{a[>0]} (m + k - 1, p). \end{aligned}$$

When $c = +1$, the $1 \bmod k$ is only useful when $k = 1$ to guarantee that we remain between 0 and $m + k - 1 = m$.

- When we are between $m + 1$ and $m + k - 2$, we simply copy the transitions from the mappings τ_ℓ. We also know that the counter is necessarily non-zero. For all $\ell \in \{m + 1, \dots, m + k - 2\}$, let $(p, c) = \tau_\ell(q, a)$ and define

$$(\ell, q) \xrightarrow[c]{a[>0]} (\ell + c, p).$$

- Finally, it remains to treat the last level in the periodic part, *i. e.*, the level $m + k - 1$. Observe that when $k = 1$, this case is already handled by the second item (as $m + k - 1 = m$). So, assume $k > 1$ and let $(p, c) = \tau_{m+k-1}(q, a)$ and define the transitions as follows:

$$\begin{aligned} c = 0 \Rightarrow (m + k - 1, q) &\xrightarrow[0]{a[>0]} (m + k - 1, p) \\ c = +1 \Rightarrow (m + k - 1, q) &\xrightarrow[+1]{a[>0]} (m, p) \\ c = -1 \Rightarrow (m + k - 1, q) &\xrightarrow[-1]{a[>0]} (m + k - 2, p). \end{aligned}$$

Clearly, the size of $\mathcal{A}_\alpha$ is polynomial in m, k and K. It remains to show that $\mathcal{L}(\mathcal{A}_\alpha) = L = \mathcal{L}(\mathcal{A})$ to obtain Proposition 5.3.12, *i. e.*,

$$\forall u \in \Sigma^*, u \in L \Leftrightarrow u \in \mathcal{L}(\mathcal{A}_\alpha).$$

To do so, given how are defined the final states of $\mathcal{A}_\alpha$, we just need the following lemma, stating the equivalence between the runs of $BG(\mathcal{A})$ and the counted runs of $\mathcal{A}_\alpha$.

Lemma A.2.1. *We have* $[\![\varepsilon]\!]_{\sim_\mathcal{A}} \xrightarrow{u} [\![u]\!]_{\sim_\mathcal{A}} \in$ *runs*$(BG(\mathcal{A}))$ *if and only if*

$$((0, \nu_0([\![\varepsilon]\!]_{\sim_\mathcal{A}})), 0) \xrightarrow{u} ((c, \nu_c([\![u]\!]_{\sim_\mathcal{A}})), \max\{0, n - m\})$$

is a counted run of $\mathcal{A}_\alpha$, with $n = cv^{\mathcal{A}}(u)$ and

$$c = \begin{cases} n & \text{if } n \leq m, \\ m + ((n-m) \bmod k) & \text{otherwise.} \end{cases}$$

Notice that if $[\![u]\!]_{\sim_\mathcal{A}}$ is a reachable state in $BG(\mathcal{A})$, then $u \in \mathit{Pref}(L)$ by definition. Moreover, by construction of $BG(\mathcal{A})$, only the reachable states $[\![u]\!]_{\sim_\mathcal{A}}$ are present in $BG(\mathcal{A})$. Thus, $u \in \mathit{Pref}(L)$ and $cv^{\mathcal{A}}(u)$ is well-defined.

Proof of Lemma A.2.1. We do a proof by induction over the length of u.

Base case. Let $u \in \Sigma^*$ such that $|u| = 0$, *i.e.*, $u = \varepsilon$. We have

$$(q_0^{\mathcal{A}_\alpha}, 0) \xrightarrow{\varepsilon} (q_0^{\mathcal{A}_\alpha}, 0) \in \mathit{cruns}(\mathcal{A}_\alpha)$$
$$q_0^{BG(\mathcal{A})} \xrightarrow{\varepsilon} q_0^{BG(\mathcal{A})} \in \mathit{runs}(BG(\mathcal{A}))$$

and

$$q_0^{\mathcal{A}_\alpha} = \nu_0([\![\varepsilon]\!]_{\sim_\mathcal{A}}).$$

Hence, the equivalence between the run and the counted run is satisfied.

Induction step. Let $i \in \mathbb{N}$ and assume the lemma holds for any $v \in \Sigma^*$ of length i. Let $u \in \Sigma^*$ such that $|u| = i + 1$, *i.e.*, $u = va$ with $a \in \Sigma$ and $v \in \Sigma^*$ such that $|v| = i$. By the induction hypothesis, we thus have $[\![\varepsilon]\!]_{\sim_\mathcal{A}} \xrightarrow{v} [\![v]\!]_{\sim_\mathcal{A}} \in \mathit{runs}(BG(\mathcal{A}))$ if and only if

$$((0, \nu_0([\![\varepsilon]\!]_{\sim_\mathcal{A}})), 0) \xrightarrow{v} ((c', \nu_{c'}([\![v]\!]_{\sim_\mathcal{A}})), \max\{0, n' - m\})$$

is a counted run of $\mathcal{A}_\alpha$, with $n' = cv^{\mathcal{A}}(v)$ and

$$c' = \begin{cases} n' & \text{if } n' \leq m, \\ m + ((n'-m) \bmod k) & \text{otherwise.} \end{cases}$$

It is consequently sufficient to prove that the last transition is correct, *i.e.*, $[\![v]\!]_{\sim_\mathcal{A}} \xrightarrow{a} [\![u]\!]_{\sim_\mathcal{A}} \in \mathit{runs}(BG(\mathcal{A}))$ if and only if

$$((c', \nu_{c'}([\![v]\!]_{\sim_\mathcal{A}})), \max\{n' - m\}) \xrightarrow{a} ((c, \nu_c([\![u]\!]_{\sim_\mathcal{A}})), \max\{n - m\})$$

is a counted run of $\mathcal{A}_\alpha$, with $n = cv^{\mathcal{A}}(u)$ and

$$c = \begin{cases} n & \text{if } n \leq m, \\ m + ((n-m) \quad \bmod k) & \text{otherwise.} \end{cases}$$

We do so in three cases:

- If $(n' < m) \lor (n' = m \land n \leq m)$, then $\max\{0, n - m\} = 0 = \max\{0, n' - m\}$, *i.e.*, we only have to use the guard $=0$ in the transitions of $\mathcal{A}_\alpha$. Moreover, we do not change the counter value. We have

$c' = n' = cv^{\mathcal{A}}(v), c = n = cv^{\mathcal{A}}(u)$, and the following equivalences:[3]

3: Note that $-1 \le n - n' \le 1$.

$$\begin{aligned}
& [\![v]\!]_{\sim_{\mathcal{A}}} \xrightarrow{a} [\![u]\!]_{\sim_{\mathcal{A}}} \in \mathit{runs}(BG(\mathcal{A})) \\
\Leftrightarrow\quad & \tau_{n'}(\nu_{n'}([\![v]\!]_{\sim_{\mathcal{A}}}), a) = (\nu_n([\![u]\!]_{\sim_{\mathcal{A}}}), n - n') \\
\Leftrightarrow\quad & \tau_{c'}(\nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}}), a) = (\nu_c([\![u]\!]_{\sim_{\mathcal{A}}}), n - n') \\
\Leftrightarrow\quad & (c', \nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}})) \xrightarrow[0]{a[=0]} (c, \nu_c([\![u]\!]_{\sim_{\mathcal{A}}})) \in \mathit{runs}(\mathcal{A}_\alpha) \\
\Leftrightarrow\quad & ((c', \nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}})), 0) \xrightarrow{a} ((c, \nu_c([\![u]\!]_{\sim_{\mathcal{A}}})), 0) \\
& \text{is a counted run of } \mathcal{A}_\alpha.
\end{aligned}$$

- If $n' = m$ and $n > m$, then $n = m + 1$ since $n' = cv^{\mathcal{A}}(v)$ and $n = cv^{\mathcal{A}}(u)$.[4] Moreover, $\max\{0, n' - m\} = 0$ and $\max\{0, n - m\} = \max\{0, m + 1 - m\} = 1$. Again, we only need to use the guard $=0$. However, we increase the counter value this time. We have $c' = m, c = m + ((m + 1 - m) \bmod k) = m + (1 \bmod k)$, and the following equivalences:

4: So, $n - n' = 1$.

$$\begin{aligned}
& [\![v]\!]_{\sim_{\mathcal{A}}} \xrightarrow{a} [\![u]\!]_{\sim_{\mathcal{A}}} \in \mathit{runs}(BG(\mathcal{A})) \\
\Leftrightarrow\quad & \tau_{n'}(\nu_{n'}([\![v]\!]_{\sim_{\mathcal{A}}}), a) = (\nu_n([\![u]\!]_{\sim_{\mathcal{A}}}), n - n') \\
\Leftrightarrow\quad & \tau_{c'}(\nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}}), a) = (\nu_c([\![u]\!]_{\sim_{\mathcal{A}}}), +1) \\
\Leftrightarrow\quad & (c', \nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}})) \xrightarrow[+1]{a[=0]} (c, \nu_c([\![u]\!]_{\sim_{\mathcal{A}}})) \in \mathit{runs}(\mathcal{A}_\alpha) \\
\Leftrightarrow\quad & ((c', \nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}})), 0) \xrightarrow{a} ((c, \nu_c([\![u]\!]_{\sim_{\mathcal{A}}})), 1) \\
& \text{is a counted run of } \mathcal{A}_\alpha.
\end{aligned}$$

- Finally, if $n' > m$, then $\max\{0, n' - m\} = n' - m$ and $\max\{0, n - m\} = n - m$. This time, we modify the counter value and we need to use the guard >0. We have $c' = m + ((n' - m) \bmod k), c = m + ((n - m) \bmod k)$, and the following equivalences:[5]

5: Again, $-1 \le n - n' \le 1$.

$$\begin{aligned}
& [\![v]\!]_{\sim_{\mathcal{A}}} \xrightarrow{a} [\![u]\!]_{\sim_{\mathcal{A}}} \in \mathit{runs}(BG(\mathcal{A})) \\
\Leftrightarrow\quad & \tau_{n'}(\nu_{n'}([\![v]\!]_{\sim_{\mathcal{A}}}), a) = (\nu_n([\![u]\!]_{\sim_{\mathcal{A}}}), n - n') \\
\Leftrightarrow\quad & \tau_{c'}(\nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}}), a) = (\nu_c([\![u]\!]_{\sim_{\mathcal{A}}}), n - n') \\
\Leftrightarrow\quad & (c', \nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}})) \xrightarrow[n-n']{a[>0]} (c, \nu_c([\![u]\!]_{\sim_{\mathcal{A}}})) \in \mathit{runs}(\mathcal{A}_\alpha) \\
\Leftrightarrow\quad & ((\nu_{c'}([\![v]\!]_{\sim_{\mathcal{A}}}), c'), n' - m) \xrightarrow{a} ((\nu_c([\![u]\!]_{\sim_{\mathcal{A}}}), c), n - m) \\
& \text{is a counted run of } \mathcal{A}_\alpha.
\end{aligned}$$

We have shown that the last transition is correct, in every case. □

By this lemma, it is thus clear that $\mathcal{L}(\mathcal{A}_\alpha) = L = \mathcal{L}(BG(\mathcal{A}))$. That is, we have showed Proposition 5.3.12 when $k \neq 0$. We thus covered every case.

A.3. Proof of Proposition 5.4.13

Proposition 5.4.13. *Let $\mathcal{O}_{\leq \ell}$ be an observation table up to $\ell \in \mathbb{N}$. Then,*

$$\forall u, v \in \overline{R \cup R\Sigma} : u \sim_{\mathcal{A}} v \Rightarrow u \in \mathit{Approx}(v).$$
$$\forall u, v \in R \cup R\Sigma \setminus \overline{R \cup R\Sigma} : \mathit{Approx}(u) = \mathit{Approx}(v).$$

Let $u, v \in \overline{R \cup R\Sigma}$ be such that $u \sim_{\mathcal{A}} v$. We show that $u \in \mathit{Approx}(v)$. That is,

$$\forall s \in S : T(us) = T(vs) \tag{A.3.i}$$
$$\forall s \in S : C(us) \neq \perp \wedge C(vs) \neq \perp \Rightarrow C(us) = C(vs) \tag{A.3.ii}$$

Proving (A.3.i). We start by proving (A.3.i). Let $s \in S$ be such that $us \in L_{\leq \ell}$. We show that $vs \in L_{\leq \ell}$. Since $u \sim_{\mathcal{A}} v$ and $us \in L_{\leq \ell} \subseteq L$, it follows that $vs \in L$. In order to conclude $vs \in L_{\leq \ell}$, we need to prove that $\forall x \in \mathit{Pref}(vs)$, we have $\mathit{cv}^{\mathcal{A}}(x) \leq \ell$. Let $x \in \mathit{Pref}(vs)$.

- Assume $x \in \mathit{Pref}(v)$. By hypothesis, we know that v is not a $\perp$-word, *i.e.*, $v \in \overline{R \cup R\Sigma}$. Thus, $v \in \mathit{Pref}(\mathcal{O}_{\leq \ell}) \subseteq \mathit{Pref}(L_{\leq \ell})$, allowing us to conclude that $\mathit{cv}^{\mathcal{A}}(x) \leq \ell$.
- Assume $x = vy$ with $y \in \mathit{Pref}(s)$. As $u \sim_{\mathcal{A}} v$ and $uy, vy \in \mathit{Pref}(L)$ (since $us, vs \in L$), we obtain that $\mathit{cv}^{\mathcal{A}}(uy) = \mathit{cv}^{\mathcal{A}}(vy)$. As $us \in L_{\leq \ell}$ by hypothesis, we know that $\mathit{cv}^{\mathcal{A}}(uy) \leq \ell$. Therefore, $\mathit{cv}^{\mathcal{A}}(x) = \mathit{cv}^{\mathcal{A}}(vy) \leq \ell$.

We have proved $us \in L_{\leq \ell} \Rightarrow vs \in L_{\leq \ell}$. The other implication is proved similarly. We thus conclude that $T(us) = T(vs)$, as us and vs are either both in $L_{\leq \ell}$, or both not in the language.

Proving (A.3.ii). We now prove (A.3.ii). Let $s \in S$ be such that $C(us) \neq \perp$ and $C(vs) \neq \perp$. By hypothesis, we have $us, vs \in \mathit{Pref}(\mathcal{O}_{\leq \ell}) \subseteq \mathit{Pref}(L_{\leq \ell})$ and, thus, $us, vs \in \mathit{Pref}(L)$. Since $u \sim_{\mathcal{A}} v$, it holds $\mathit{cv}^{\mathcal{A}}(us) = \mathit{cv}^{\mathcal{A}}(vs) \leq \ell$. Moreover, since $us, vs \in \mathit{Pref}(\mathcal{O}_{\leq \ell})$, we have $C(us) = \mathit{cv}^{\mathcal{A}}(us) = \mathit{cv}^{\mathcal{A}}(vs) = C(vs)$, which proves (A.3.ii).

Second part of the proposition. We now proceed to the proof of the second part of the proposition. Any word $u \in R \cup R\Sigma \setminus \overline{R \cup R\Sigma}$ is a $\perp$-word. It follows that for all $s \in S$, $T(us) =$ **no** and $C(us) = \perp$. Therefore all the $\perp$-words are in the same approximation set.

A.4. Proof of Lemma 5.4.19

Lemma 5.4.19. *Let $\mathcal{O}_{\leq \ell} = (R, S, \hat{S}, T, C)$ be an observation table and $\mathcal{O}'_{\leq \ell} = (R', S', \hat{S}', T', C')$ be the observation table obtained after resolving a u-openness (with $u \in R\Sigma \setminus R$). Then,*

- $|R'| = |R| + 1$, $|S'| = |S|$, $\left|\hat{S}'\right| = \left|\hat{S}\right|$,
- *Approx*$'(u) \cap R' = \{u\}$, *and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in* $|\mathcal{O}_{\leq \ell}|$.

Recall that we resolve a u-openness by "promoting" u (currently in $R\Sigma \setminus R$) to the set of representatives. This means that there are new rows in $R\Sigma$. Observe $S, \hat{S}$ are left unchanged.

The first item of the lemma is easily proved, given how a openness is resolved. We prove the remaining two items of the lemma in order.

u is the unique element in *Approx*$'(u)$**.** Since $R' = R \cup \{u\}$ and $u \in$ *Approx*$'(u)$, it suffices to show that *Approx*$'(u) \cap R = \emptyset$ to get *Approx*$'(u) \cap R' = \{u\}$.

By contradiction, assume $\exists v \in R$ such that $v \in$ *Approx*$'(u)$. By Lemma 5.4.12, since $v \in R$ and $u \in R\Sigma$, we have $v \in$ *Approx*(u). However, due to the u-openness, we know that *Approx*$(u) \cap R = \emptyset$. We thus have a contradiction and we conclude that *Approx*$'(u) \cap R' = \{u\}$.

Lemma 5.4.12. Let $\mathcal{O}_{\leq \ell}$ and $\mathcal{O}'_{\leq \ell}$ be two observation tables up to the same counter limit $\ell \in \mathbb{N}$ such that $R \cup R\Sigma \subseteq R' \cup R'\Sigma, S \subseteq S'$, and $\hat{S} \subseteq \hat{S}'$. Then, for all $u, v \in R \cup R\Sigma$ such that $u \in$ *Approx*$'(v)$, we have $u \in$ *Approx*(v).

Number of queries. Let us prove the last item. To extend and update $\mathcal{O}_{\leq \ell}$, the learner asks queries to the teacher in order to compute the new value $T'(uas)$ (resp. $C'(uas)$) for each $a \in \Sigma$ and each $s \in \hat{S}$ (resp. $s \in S$). Moreover, as $R' \neq R$, the set *Pref*$(\mathcal{O}'_{\leq \ell})$ may strictly include *Pref*$(\mathcal{O}_{\leq \ell})$, and thus all the values $C'(u's)$, $u' \in R \cup R\Sigma$, $s \in S$ must be recomputed. By Lemma 5.4.7, this requires a number of membership and counter value queries polynomial in $|\mathcal{O}_{\leq \ell}|$.

Lemma 5.4.7. Filling T and C requires a polynomial number of **MQ** and **CVQ** in the sizes of $R \cup R\Sigma$ and $\hat{S}$.

A.5. Proof of Lemma 5.4.20

Lemma 5.4.20. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and $\mathcal{O}'_{\leq \ell}$ be the observation table obtained after resolving a (u, v, a)-Σ-inconsistency (with $ua \in R\Sigma$ and $v \in R$). Then,*

- $|R'| = |R|$, $|S'| = |S| + 1$, $\left|\hat{S}'\right| = \left|\hat{S}\right| + 1$,
- $v \notin$ *Approx*$'(u)$,
- $\left|\textit{Approx}'(u)\right| < \left|\textit{Approx}(u)\right|$, *and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in* $|\mathcal{O}_{\leq \ell}|$.

Recall that we resolve a (u, v, a)-Σ-inconsistency as follows. By definition, there exists $s \in S$ such that either

- $T(uas) \neq T(vas)$, or
- $C(uas) \neq \bot \land C(vas) \neq \bot \land C(uas) \neq C(vas)$.

In both cases, we add as as a new separator in both S and $\hat{S}$. Observe that R is left unchanged.

The first item of the lemma is easily proved, given the above procedure. We prove the remaining three items of the lemma in order.

v is no longer in the approximation set of u. Let us first explain why $v \notin \mathit{Approx}'(u)$. Let $as \in S'$ as above. We have two cases.

- If $T(uas) \neq T(vas)$, we keep the same inequality $T'(uas) \neq T'(vas)$, as ℓ is unchanged. Thus, by definition, $v \notin \mathit{Approx}'(u)$.
- If $C(uas) \neq \bot$, $C(vas) \neq \bot$, and $C(uas) \neq C(vas)$, we have $C'(uas) = C(uas) \neq \bot$ (likewise for vas), Therefore, it holds that $C'(uas) \neq \bot$, $C'(vas) \neq \bot$, and $C'(uas) \neq C'(vas)$. Thus, we again get $v \notin \mathit{Approx}'(u)$.

The approximation set of u is smaller. As $v \in \mathit{Approx}(u)$, $v \notin \mathit{Approx}'(u)$, and $R = R'$, we get $|\mathit{Approx}'(u)| < |\mathit{Approx}(u)|$ by Lemma 5.4.12.

Number of queries. Let us prove the last item. To extend and update $\mathcal{O}_{\leq \ell}$, the learner asks queries to the teacher so as to compute the new value $T'(u'as)$ (resp. $C'(u'as)$) for each $u' \in R \cup R\Sigma$. Moreover, as $\mathit{Pref}(\mathcal{O}'_{\leq \ell})$ may have changed, we must recompute the values $C'(u's')$ for every $u' \in R \cup R\Sigma$ and $s' \in S$. By Lemma 5.4.7, this requires a number of membership and counter value queries in polynomial $|\mathcal{O}_{\leq \ell}|$.

Lemma 5.4.12. Let $\mathcal{O}_{\leq \ell}$ and $\mathcal{O}'_{\leq \ell}$ be two observation tables up to the same counter limit $\ell \in \mathbb{N}$ such that $R \cup R\Sigma \subseteq R' \cup R'\Sigma, S \subseteq S'$, and $\hat{S} \subseteq \hat{S}'$. Then, for all $u, v \in R \cup R\Sigma$ such that $u \in \mathit{Approx}'(v)$, we have $u \in \mathit{Approx}(v)$.

Lemma 5.4.7. Filling T and C requires a polynomial number of **MQ** and **CVQ** in the sizes of $R \cup R\Sigma$ and $\hat{S}$.

A.6. Proof of Lemma 5.4.21

Lemma 5.4.21. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and $\mathcal{O}'_{\leq \ell}$ be the observation table obtained after resolving a (u, v, s)-$\perp$-inconsistency (with $u, v \in R \cup R\Sigma$ and $s \in S$). Then,*

- $|R'| = |R|$, $|S'| \leq |R \cup R\Sigma| + \left|\hat{S}\right|$, $\left|\hat{S}'\right| \leq |R \cup R\Sigma| + \left|\hat{S}\right|$,
- *if u' is a prefix of u, then $u \notin$ Approx$'(v)$,*
- *if u is a proper prefix of u', then,*
  - *if $vs'' \in L_{\leq \ell}$, then $u \in$ Approx$'(v)$ implies that $C'(us) = C'(vs)$, and*
  - *if $vs'' \notin L_{\leq \ell}$, then $u \notin$ Approx$'(v)$,*
- *either $|$Approx$'(v)| < |$Approx$(v)|$ or the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated,*
- *if the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated, we have $|S'| = |S|$, and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in $|\mathcal{O}_{\leq \ell}|$.*

Let us recall how we resolve a (u, v, s)-$\perp$-inconsistency. Let us assume, without loss of generality, that $C(us) \neq \perp$ and $C(vs) = \perp$. So, $us \in$ *Pref*$(\mathcal{O}_{\leq \ell})$, *i.e.*, there exist $u' \in R \cup R\Sigma$ and $s' \in \hat{S}$ such that $us \in$ *Pref*$(u's')$ and $T(u's') =$ **yes**. We denote by s'' the word such that $us'' = u's'$. Notice that s is a prefix of s''. We have two cases:

- If u' is a prefix of u, we add all suffixes of s'' to S and leave R and $\hat{S}$ unchanged.
- If u is a proper prefix of u', there are again two cases:
  - If $vs'' \in L_{\leq \ell}$, we add all suffixes of s'' to $\hat{S}$ and leave R and S unchanged.
  - If $vs'' \notin L_{\leq \ell}$, we all all suffixes of s'' to S and $\hat{S}$, and leave R unchanged.

We start by proving the second, third, and fourth items of the lemma, before showing the first and the last two items.[6]

6: Unlike in the last two sections, the number of representatives and separators is not immediate.

If u' is a prefix of u, then $u \notin$ *Approx*$'(v)$. Suppose u' is a prefix of u, and consider $s'' \in \hat{S}$. We have $T(vs'') =$ **no** since otherwise, as explained above, $vs \in$ *Pref*$(\mathcal{O}_{\leq \ell})$ in contradiction with $C(vs) = \perp$. Therefore, $T'(vs'') = T(vs'') =$ **no**. We also have $T'(us'') = T'(u's') = T(u's') =$ **yes** (by definition of $u's'$). Since $s'' \in S'$, it follows that $u \notin$ *Approx*$'(v)$.

When u is a proper prefix of u'. Suppose that u is a proper prefix of u'. The subcase $vs'' \notin L_{\leq \ell}$ is solved similarly. Consider again $s'' \in S'$. By hypothesis we have $T'(vs'') =$ **no**. Recall that we have $T'(us'') =$ **yes**. Therefore $u \notin$ *Approx*$'(v)$.

In the other subcase ($vs'' \in L_{\leq \ell}$), we have $vs \in$ *Pref*$(\mathcal{O}'_{\leq \ell})$ since $s'' \in \hat{S}'$, $T'(vs'') =$ **yes** and vs is a prefix of vs''. As $us \in$ *Pref*$(\mathcal{O})$ by hypothesis, we also have $us \in \mathit{Pref}(\mathcal{O}'_{\leq\ell})$. Hence $C'(us) \neq \bot \wedge C'(vs) \neq \bot$. If $v \in \mathit{Approx}'(u)$, it follows that $C'(us) = C'(vs)$.

The approximation set is smaller or a mismatch is eliminated. The fourth item of the lemma follows directly from the first two items, thanks to Lemma 5.4.12.

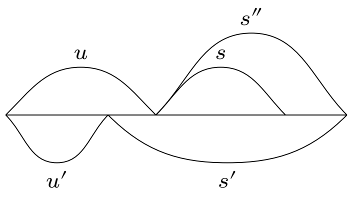
Lemma 5.4.12. Let $\mathcal{O}_{\leq\ell}$ and $\mathcal{O}'_{\leq\ell}$ be two observation tables up to the same counter limit $\ell \in \mathbb{N}$ such that $R \cup R\Sigma \subseteq R' \cup R'\Sigma, S \subseteq S'$, and $\widehat{S} \subseteq \widehat{S}'$. Then, for all $u, v \in R \cup R\Sigma$ such that $u \in \mathit{Approx}'(v)$, we have $u \in \mathit{Approx}(v)$.

The first and last two items. We consider the different cases as above, which all together will lead to the remaining items of this lemma.

Suppose first that u' is a prefix of u. Then by construction of $\mathcal{O}'_{\leq\ell}$, we have $R' = R$, $S' = S \cup \mathit{Suff}(s'')$, and $\widehat{S}' = \widehat{S}$. As $S' \subseteq \widehat{S}'$, we get $|S'| \leq \left|\widehat{S}\right|$. Moreover, as R, $\widehat{S}$ are unchanged and $\mathit{Suff}(s'')$ is added to S, we only need to add the new values $C'(yz)$ to $\mathcal{O}'_{\leq\ell}$, for all $y \in R \cup R\Sigma$ and $z \in \mathit{Suff}(s'')$. Therefore there is no **MQ** and a polynomial number of **CVQ** in $|\mathcal{O}_{\leq\ell}|$ (by Lemma 5.4.7).

Lemma 5.4.7. Filling T and C requires a polynomial number of **MQ** and **CVQ** in the sizes of $R \cup R\Sigma$ and $\widehat{S}$.

Second, suppose that u is a proper prefix of u'. We have two subcases according to whether $vs'' \in L_{\leq\ell}$. To determine which subcase applies, by Lemma 5.4.7 and the preceding argument, this requires one **MQ** and at most $|vs''|$ **CVQ**. The latter number is polynomial in the size of $\mathcal{O}_{\leq\ell}$. Indeed we have $v \in R \cup R\Sigma$, $s'' = xs'$ with $s' \in \widehat{S}$ and x prefix of $u' \in R \cup R\Sigma$ (see Figure 5.8a, which is repeated in the margin), and thus



$$|vs''| \leq |v| + |u'| + |s'|.$$

Consider the first subcase $vs'' \in L_{\leq\ell}$. Then by construction of $\mathcal{O}'_{\leq\ell}$, we have $R' = R$, $S' = S$, and $\widehat{S}' = \widehat{S} \cup \mathit{Suff}(s'')$. Recall that as $s' \in \widehat{S}$ is a suffix of s'', there are only $|x|$ suffixes to add to $\widehat{S}$ with $|x| \leq |u'|$. Thus,

$$\left|\widehat{S}'\right| \leq \left|\widehat{S}\right| + |R \cup R\Sigma|.$$

Notice also that this is the only subcase where the mismatch $C(us) \neq \bot \Leftrightarrow C(vs) = \bot$ may be eliminated and for which we have $|S'| = |S|$. For each new suffix z to add to $\widehat{S}$ and each $y \in R \cup R\Sigma$, we have to add the new value $T'(yz)$ to $\mathcal{O}'_{\leq\ell}$, and this requires a polynomial number of **MQ** and **CVQ** in $|\mathcal{O}_{\leq\ell}|$ (by Lemma 5.4.7). Moreover, as $\mathit{Pref}(\mathcal{O}'_{\leq\ell})$ may have changed, the values $C'(yz)$ have to be recomputed for all $y \in R \cup R\Sigma$, $z \in S$. This requires again a polynomial number of **CVQ** in $|\mathcal{O}_{\leq\ell}|$.

Finally consider the second subcase $vs'' \notin L_{\leq\ell}$. Then by construction of $\mathcal{O}'_{\leq\ell}$, we have $R' = R$, $S' = S \cup \mathit{Suff}(s'')$, and $\widehat{S}' = \widehat{S} \cup \mathit{Suff}(s'')$. Therefore this case is similar to the previous subcase with the exception that $S' = S \cup \mathit{Suff}(s'')$. The same arguments can be repeated for $\widehat{S}$. As $S' \subseteq \widehat{S}'$, it follows that

$$|S'| \leq \left|\widehat{S}\right| + |R \cup R\Sigma|.$$

The way that $\mathcal{O}'_{\leq\ell}$ is extended and updated can also be repeated, which leads again to a polynomial number of **MQ** and **CVQ**.

A.7. Proof of Proposition 5.4.24

Proposition 5.4.24. *Let $BG(\mathcal{A})$ be the behavior graph of an ROCA $\mathcal{A}$, K be its width, m, k be the offset and the period of a periodic description of $BG(\mathcal{A})$, $s = m + (K \cdot k)^4$, and $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\bot$-consistent observation table up to $\ell > s$ such that $\mathcal{L}(\mathcal{H}_{\leq \ell}) = L_{\leq \ell}$. Then, the subgraphs of $BG(\mathcal{A})$ and $\mathcal{H}_{\leq \ell}$ restricted to the reachable and co-reachable states and to the levels 0 to $\ell - s$ are isomorphic.*

Proving this proposition requires a preliminary lemma stating that one can bound the height $cv^{\mathcal{A}}(w)$ of a witness w for non-equivalence with respect to $\sim_{\mathcal{A}}$. We recall that $height^{\mathcal{A}}(w) = \max_{x \in Pref(w)} cv^{\mathcal{A}}(x)$.

Lemma A.7.1. *Let $\mathcal{A}$ be an ROCA accepting a language $L \subseteq \Sigma^*$, $BG(\mathcal{A})$ be its behavior graph, α be a periodic description of $BG(\mathcal{A})$ with offset m and period k, and $s = m + (K \cdot k)^4$. Let $[\![u]\!]_{\sim_{\mathcal{A}}}$ and $[\![v]\!]_{\sim_{\mathcal{A}}}$ be two distinct states of $BG(\mathcal{A})$ such that $cv^{\mathcal{A}}(u) = cv^{\mathcal{A}}(v)$. Then, there exists a word $w \in \Sigma^*$ such that*

$$uw \in L \Leftrightarrow vw \notin L,$$
$$height^{\mathcal{A}}(uw) \leq s + cv^{\mathcal{A}}(u),$$

and

$$height^{\mathcal{A}}(vw) \leq s + cv^{\mathcal{A}}(u)$$

Such a property is proved in [NL10, Lemma 3] for VOCAs and immediately transfers to ROCAs by Theorem 5.3.10. We now prove Proposition 5.4.24.

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

Theorem 5.3.10. Let $\mathcal{A}$ be a sound ROCA, $BG(\mathcal{A})$ be its behavior graph, $\widetilde{\mathcal{A}}$ be the corresponding VOCA accepting $\widetilde{L}$, and $BG(\widetilde{L})$ be the behavior graph of $\widetilde{L}$. Then,

- $BG(\mathcal{A})$ and $BG(\widetilde{L})$ are isomorphic up to $\lambda_{\mathcal{A}}$ and $\widetilde{\lambda}$, and
- the isomorphism respects the counter values (*i.e.*, level membership) and both offset and period of periodic descriptions.

Proof of Proposition 5.4.24. Let $BG(\mathcal{A})$ be the behavior graph of $\mathcal{A}$ and let $\mathcal{H}_{\leq \ell}$ be the DFA constructed from the table $\mathcal{O}_{\leq \ell}$. Recall that by definition of $\overline{R}$, $u \in \overline{R} \Leftrightarrow C(u) \neq \bot$, *i.e.*, $u \in Pref(\mathcal{O}_{\leq \ell})$. Recall that $Pref(\mathcal{O}_{\leq \ell}) \subseteq Pref(L)$. We denote by $\overline{R}_{\leq \ell - s}$ (resp. $\overline{R \cup R\Sigma}_{\leq \ell - s}$) the set of elements u of $\overline{R}$ (resp. $\overline{R \cup R\Sigma}$) such that $height^{\mathcal{A}}(u) \leq \ell - s$.
We consider the following two subautomata:

- $\mathcal{B}$ equal to $\mathcal{H}_{\leq \ell}$ restricted to the states $[\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}}$ with $u \in \overline{R}_{\leq \ell - s}$, and
- the bounded behavior graph $BG_{\leq \ell - s}(\mathcal{A})$.

Both automata accept $L_{\leq \ell - s}$ and all their states are reachable. We show that there exists an isomorphism between the two. Let φ such that

$$\forall u \in \overline{R}_{\leq \ell - s} : \varphi([\![u]\!]_{\equiv_{\mathcal{O}_{\leq \ell}}}) = [\![u]\!]_{\sim_{\mathcal{A}}}.$$

We prove that φ embeds $\mathcal{B}$ to $BG_{\leq \ell - s}(\mathcal{A})$. Therefore $\mathcal{B}$ and $\varphi(\mathcal{B})$ will be isomorphic subautomata both accepting $L_{\leq \ell - s}$. It will follow that the reachable and co-reachable parts of $\mathcal{B}$ and $BG_{\leq \ell - s}(\mathcal{A})$ will also be isomorphic as stated in Proposition 5.4.24.
First notice that by definition of φ, $[\![u]\!]_{\sim_{\mathcal{A}}}$ is a state of $BG_{\leq \ell - s}(\mathcal{A})$ because $u \in Pref(L)$ and $height^{\mathcal{A}}(u) \leq \ell - s$, for all $u \in \overline{R}_{\leq \ell - s}$.

Then let us show that:

$$\forall u, v \in \overline{R \cup R\Sigma}_{\leq \ell - s} : u \equiv_{\mathcal{O}_{\leq \ell}} v \Leftrightarrow u \sim_{\mathcal{A}} v, \tag{A.7.i}$$

in order to deduce that φ respects the equivalence classes. Let $u, v \in \overline{R \cup R\Sigma}_{\leq \ell - s}$. By Proposition 5.4.13, we already know that $u \sim_{\mathcal{A}} v \Rightarrow u \equiv_{\mathcal{O}_{\leq \ell}} v$. Assume $u \equiv_{\mathcal{O}_{\leq \ell}} v$ and, towards a contradiction, assume $u \nsim_{\mathcal{A}} v$. If $\mathit{cv}^{\mathcal{A}}(u) \neq \mathit{cv}^{\mathcal{A}}(v)$, it holds that $C(u) \neq C(v)$ (since $u, v \in \overline{R \cup R\Sigma}$). It follows that $u \notin \mathit{Approx}(v)$, that is, $u \not\equiv_{\mathcal{O}_{\leq \ell}} v$, which is a contradiction. So, we get $\mathit{cv}^{\mathcal{A}}(u) = \mathit{cv}^{\mathcal{A}}(v)$. Since $u \nsim_{\mathcal{A}} v$, we know by Lemma A.7.1 that there exists a witness $w \in \Sigma^*$ such that $uw \in L \Leftrightarrow vw \notin L$ and $\mathit{height}^{\mathcal{A}}(uw), \mathit{height}^{\mathcal{A}}(vw) \leq \ell$ (since $s + \mathit{cv}^{\mathcal{A}}(u) \leq s + \ell - s = \ell$). Since $\mathcal{L}(\mathcal{H}_{\leq \mathcal{A}}) = L_{\leq \ell}$, it is impossible that

$$q_0^{\mathcal{H}_{\leq \ell}} \xrightarrow{u} p \in \mathit{runs}(\mathcal{H}_{\leq \ell}) \qquad \text{and} \qquad q_0^{\mathcal{H}_{\leq \ell}} \xrightarrow{v} p \in \mathit{runs}(\mathcal{H}_{\leq \ell}).$$

That is, both words cannot end in the same state. As $u, v \in R \cup R\Sigma$, by definition of $\mathcal{H}_{\leq \ell}$, it follows that $u \not\equiv_{\mathcal{O}_{\leq \ell}} v$ which is again a contradiction. We have proved that $u \equiv_{\mathcal{O}_{\leq \ell}} v \Leftrightarrow u \sim_{\mathcal{A}} v$.
Note that φ puts the initial states of both subautomata, $\mathcal{B}$ and $BG_{\leq \ell - s}(\mathcal{A})$, in correspondence:

$$\begin{aligned}\varphi(q_0^{\mathcal{B}}) = \varphi(q_0^{\mathcal{H}_{\leq \ell}}) = \varphi(\llbracket \varepsilon \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}}) = \llbracket \varepsilon \rrbracket_{\sim_{\mathcal{A}}} &= q_0^{BG_{\leq \ell}(\mathcal{A})} \\ &= q_0^{BG_{\leq \ell - s}(\mathcal{A})}.\end{aligned}$$

Let us show that it is also the case for the final states. Let $u \in \overline{R}_{\leq \ell - s}$. If $\llbracket u \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}} \in F^{\mathcal{B}}$, then by definition, $T(u) = \textbf{yes}$, *i.e.*, $u \in L_{\leq \ell} \subseteq L$. It follows that $\llbracket u \rrbracket_{\sim_{\mathcal{A}}} \in F^{BG_{\leq \ell - s}(\mathcal{A})}$. Conversely, if $\llbracket u \rrbracket_{\sim_{\mathcal{A}}} \in F^{BG_{\leq \ell}(\mathcal{A})}$, then by definition $u \in L$ and moreover $\mathit{height}^{\mathcal{A}}(u) \leq \ell - s \leq \ell$. Thus $u \in L_{\leq \ell}$ and $\llbracket u \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}} \in F^{\mathcal{B}}$.
It remains to prove that φ respects the transitions. Let $\llbracket u \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}}, \llbracket u' \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}} \in \overline{R}_{\leq \ell - s}$ such that $u' \equiv_{\mathcal{O}_{\leq \ell}} ua$ with $a \in \Sigma$. By (A.7.i), we have $u' \sim_{\mathcal{A}} ua$. By definition of both transition functions, we get

$$\llbracket u \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}} \xrightarrow{a} \llbracket u' \rrbracket_{\equiv_{\mathcal{O}_{\leq \ell}}} \in \mathit{runs}(\mathcal{B})$$

and

$$\llbracket u \rrbracket_{\sim_{\mathcal{A}}} \xrightarrow{a} \llbracket u' \rrbracket_{\sim_{\mathcal{A}}} \in \mathit{runs}(BG_{\leq \ell - s}(\mathcal{A})).$$

From all the above points, we conclude that φ embeds $\mathcal{B}$ in $BG_{\leq \ell - s}(\mathcal{A})$ and that it is an isomorphism between their reachable and co-reachable parts. ☐

Proposition 5.4.13. Let $\mathcal{O}_{\leq \ell}$ be an observation table up to $\ell \in \mathbb{N}$. Then, for all $u, v \in \overline{R \cup R\Sigma}$ such that $u \sim_{\mathcal{A}} v$, we have $u \in \mathit{Approx}(v)$.
Moreover, all $u \in R \cup R\Sigma \setminus \overline{R \cup R\Sigma}$ belong to the same approximation set.

A.8. Complexity of the learning algorithm

This section is devoted to the proof of our main theorem establishing the complexity of L^*_{ROCA} and the number of required queries, which we repeat for convenience.

Theorem 5.4.1. *Let $\mathcal{A}$ be the sound ROCA of the teacher and ζ be the length of the longest counterexample returned by the teacher on (partial) equivalence queries. Then,*

- *the L^*_{ROCA} algorithm eventually terminates and returns an ROCA accepting $\mathcal{L}(\mathcal{A})$ and whose size is polynomial in $|Q^\mathcal{A}|$ and $|\Sigma|$,*
- *in time and space exponential in $|Q^\mathcal{A}|, |\Sigma|$ and ζ, and*
- *asking a number of* **PEQ** *in $\mathcal{O}(\zeta^3)$, a number of* **EQ** *in $\mathcal{O}(|Q^\mathcal{A}|\zeta^2)$, and a number of* **MQ** *and* **CVQ** *exponential in $|Q^\mathcal{A}|, |\Sigma|$ and ζ.*

We already argued in Section 5.4 that it is possible to make an observation table closed, Σ- and $\bot$-consistent in finite time. More precisely, Lemmas 5.4.19 to 5.4.21 and 5.4.25 state the growth of the number of representatives and separators, and the number of needed queries, when making the table "good" and when processing the counterexamples. Let us recall these lemmas, for convenience.

Lemma 5.4.19. *Let $\mathcal{O}_{\leq \ell} = (R, S, \hat{S}, T, C)$ be an observation table and $\mathcal{O}'_{\leq \ell} = (R', S', \hat{S}', T', C')$ be the observation table obtained after resolving a u-openness (with $u \in R\Sigma \setminus R$). Then,*

- $|R'| = |R| + 1$, $|S'| = |S|$, $|\hat{S}'| = |\hat{S}|$,
- *$\mathit{Approx}'(u) \cap R' = \{u\}$, and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in $|\mathcal{O}_{\leq \ell}|$.*

Lemma 5.4.20. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and $\mathcal{O}'_{\leq \ell}$ be the observation table obtained after resolving a (u, v, a)-Σ-inconsistency (with $ua \in R\Sigma$ and $v \in R$). Then,*

- $|R'| = |R|$, $|S'| = |S| + 1$, $|\hat{S}'| = |\hat{S}| + 1$,
- *$v \notin \mathit{Approx}'(u)$,*
- *$|\mathit{Approx}'(u)| < |\mathit{Approx}(u)|$, and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in $|\mathcal{O}_{\leq \ell}|$.*

Lemma 5.4.21. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and $\mathcal{O}'_{\leq \ell}$ be the observation table obtained after resolving a (u, v, s)-$\perp$-inconsistency (with $u, v \in R \cup R\Sigma$ and $s \in S$). Then,*

- $|R'| = |R|$, $|S'| \leq |R \cup R\Sigma| + \left|\widehat{S}\right|$, $\left|\widehat{S'}\right| \leq |R \cup R\Sigma| + \left|\widehat{S}\right|$,
- *if u' is a prefix of u, then $u \notin$ Approx$'(v)$,*
- *if u is a proper prefix of u', then,*
    - *if $vs'' \in L_{\leq \ell}$, then $u \in$ Approx$'(v)$ implies that $C'(us) = C'(vs)$, and*
    - *if $vs'' \notin L_{\leq \ell}$, then $u \notin$ Approx$'(v)$,*
- *either $\left|\text{Approx}'(v)\right| < |\text{Approx}(v)|$ or the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated,*
- *if the mismatch $C(us) \neq \perp \Leftrightarrow C(vs) = \perp$ is eliminated, we have $|S'| = |S|$, and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in $|\mathcal{O}_{\leq \ell}|$.*

Lemma 5.4.25. *Let $\mathcal{O}_{\leq \ell}$ be an observation table, ζ be the length of the counterexample returned by a* **PEQ** *or* **EQ**, *and $\mathcal{O}'_{\leq \ell'}$ be the new table obtained by processing the counterexample. Then,*

- $|R'| \leq |R| + \zeta$, $|S'| = |S|$, $\left|\widehat{S'}\right| = \left|\widehat{S}\right|$, *and*
- *the number of* **MQ** *and the number of* **CVQ** *are both bounded by a polynomial in ζ and $|\mathcal{O}_{\leq \ell}|$.*

In this section, we suppose that we have an ROCA $\mathcal{A}$ that accepts a language $L \subseteq \Sigma^*$. We denote ζ the length of the longest counterexample returned by the teacher on (partial) equivalence queries. Let us call an iteration of Algorithm 5.1 a *round*. That is, a round consists in making the current table closed, Σ- and $\perp$-consistent and then handling the counterexample provided either by a **PEQ** or an **EQ**. Notice that the total number of rounds performed by the learning algorithm coincides with the total number of partial equivalence queries.

We first give a lemma giving an upper bound for values given by some recursive functions, before discussing the number of (partial) equivalence queries needed throughout the learning process. From there, we will be able to fully characterize the growth of the number of representatives and separators in Section A.8.2. Finally, Section A.8.3 gives the proof of Theorem 5.4.1.

Lemma A.8.1. *Let $\alpha, \beta \geq 1$ be constants and S be a function defined as:*

$$S(0) = \beta,$$
$$S(j) = S(j-1) \cdot \alpha \cdot j + \beta, \quad \forall j \geq 1.$$

Then, for all $j \in \mathbb{N}$, it holds that

$$S(j) \leq (j+1) \cdot (\alpha \cdot j)^j \cdot \beta.$$

Proof. Let $P(k, \ell) = k \cdot (k+1) \cdot (k+2) \cdots \ell$ for any $k, \ell \in \mathbb{N}$ such that $1 \leq k \leq \ell$. Notice that $P(k, \ell) \leq \ell^{\ell-k+1} \leq \ell^\ell$.
Let $j \in \mathbb{N}$. We have:

$$\begin{aligned}
S(j) &= S(j-1) \cdot \alpha \cdot j + \beta \\
&= (S(j-2) \cdot \alpha \cdot (j-1) + \beta) \cdot \alpha \cdot j + \beta \\
&= ((S(j-3) \cdot \alpha \cdot (j-2) + \beta) \cdot \alpha \cdot (j-1) + \beta) \cdot \alpha \cdot j + \beta \\
&= \dots \\
&= S(0) \cdot \alpha^j P(1, j) + \beta \cdot \begin{pmatrix} \alpha^{j-1} P(2, j) + \alpha^{j-2} P(3, j) \\ \qquad + \dots + \alpha P(j, j) + 1 \end{pmatrix} \\
&= \beta \cdot \begin{pmatrix} \alpha^j P(1, j) + \alpha^{j-1} P(2, j) + \alpha^{j-2} P(3, j) \\ \qquad + \dots + \alpha P(j, j) + 1 \end{pmatrix} \\
&\leq (j+1) \cdot (\alpha \cdot j)^j \cdot \beta.
\end{aligned}$$

□

A.8.1. Number of (partial) equivalence queries

We start with the number of (partial) equivalence queries asked by L^*_{ROCA}.

Proposition A.8.2. *In L^*_{ROCA},*

- *the final counter limit ℓ is bounded by ζ,*
- *the number of* **PEQ** *in in $\mathcal{O}(\zeta^3)$,*
- *the number of* **EQ** *is in $\mathcal{O}(|Q^{\mathcal{A}}|\zeta^2)$.*

Proof. The proof is inspired by [NL10]. First, notice that the counter limit ℓ of $\mathcal{O}_{\leq \ell}$ is increased only when a counterexample for an **EQ** is processed. Thus, ℓ is determined by the height of a counterexample, and this height cannot exceed $\frac{\zeta}{2} \leq \zeta$. Moreover, it follows that the number of times ℓ is increased is in $\mathcal{O}(\zeta)$.
Second, let us study the number of **PEQ** for a fixed counter limit ℓ. Recall that the index of $\equiv_{\mathcal{O}_{\leq \ell}}$ strictly increases after each provided counterexample (see Proposition 5.4.26) and that the number of equivalence classes of $\sim_{\mathcal{A}}$ up to ℓ is bounded by $(\ell+1)|Q^{\mathcal{A}}| + 1$ (see Lemma 5.4.3). By Proposition 5.4.13, it follows that we ask at most $\mathcal{O}(|Q^{\mathcal{A}}|\zeta)$ **PEQ** by fixed ℓ. Therefore, the total number of **PEQ** is $\mathcal{O}(|Q^{\mathcal{A}}|\zeta^2)$.
Finally, let us study the number of **EQ** when a DFA $\mathcal{H}_{\leq \ell}$ accepting $L_{\leq \ell}$ has been learned. We generate at most ℓ^2 periodic descriptions (as the number of pairs of offset and period is bounded by ℓ^2). We thus ask an **EQ** for the ROCA constructed from each such description, *i.e.*, at most $\mathcal{O}(\zeta^2)$ queries. This leads to a total number of **EQ** in $\mathcal{O}(\zeta^3)$. □

Proposition 5.4.26. Let $\mathcal{O}_{\leq \ell}$ be a closed, Σ- and $\perp$-consistent observation table up to $\ell \in \mathbb{N}$, and $\mathcal{O}'_{\leq \ell}$ be the closed, Σ- and $\perp$-consistent observation table obtained after processing a counterexample. Then, for all $u, v \in R \cup R\Sigma$ such that $c \equiv_{\mathcal{O}'_{\leq \ell}} v$, we have $u \equiv_{\mathcal{O}_{\leq \ell}} v$. Furthermore, the index of $\equiv_{\mathcal{O}'_{\leq \ell}}$ is strictly greater than the index of $\equiv_{\mathcal{O}_{\leq \ell}}$.

Lemma 5.4.3. The number of states of $BG_{\leq \ell}(\mathcal{A})$ is at most $(\ell+1) \cdot |Q^{\mathcal{A}}|$.

Proposition 5.4.13. Let $\mathcal{O}_{\leq \ell}$ be an observation table up to $\ell \in \mathbb{N}$. Then, for all $u, v \in R \cup R\Sigma$ such that $u \sim_{\mathcal{A}} v$, we have $u \in \textit{Approx}(v)$. Moreover, all $u \in R \cup R\Sigma \setminus R \cup R\Sigma$ belong to the same approximation set.

A.8.2. Growth of the number of representatives and separators

In this section, we study the size of the observation table at the end of the execution of our learning algorithm. We denote by R_f (resp. $\widehat{S}_f$) the final set R (resp. $\widehat{S}$) at the end of the execution. With this study, we will be able to show that making an observation table closed and consistent can be done in a finite amount of time. That is, we prove Proposition 5.4.18, which we restate. We will also be able to count the total number of **MQ** and **CVQ** performed during the execution of the algorithm.

Proposition 5.4.18. *Given an observation table $\mathcal{O}_{\leq \ell}$ up to $\ell \in \mathbb{N}$, there exists an algorithm that makes it closed, Σ- and $\perp$-consistent in a finite amount of time.*

Recall that, by Lemma 5.4.19, resolving a single openness can be done in finite time. Likewise for a single Σ-inconsistency (Lemma 5.4.20), or a single $\perp$-inconsistency (Lemma 5.4.21). We thus have to show that we apply these operations finitely many times.

We begin with the study of the size of $R_f \cup R_f\Sigma$.

Proposition A.8.3. *We have the following properties:*

- *The size of $R_f \cup R_f\Sigma$ is in $\mathcal{O}\left(|Q^{\mathcal{A}}||\Sigma|\zeta^4\right)$.*
- *During the process of making a table closed, Σ- and $\perp$-consistent, the number of resolved openness cases is in $\mathcal{O}\left(|Q^{\mathcal{A}}|\zeta\right)$.*

Proof. We first study the growth of R and the number of resolved openness cases after one round of the learning algorithm, *i.e.*, after first making the current table $\mathcal{O}_{\leq \ell}$ closed, Σ- and $\perp$-consistent, and then handling a counterexample to a (partial) equivalence query.
During the process of making $\mathcal{O}_{\leq \ell}$ closed, Σ- and $\perp$-consistent, notice that R increases only when an openness is resolved (by Lemmas 5.4.19 to 5.4.21). Furthermore, when a u-openness is resolved, R is increased with u which is the only representative in its new approximation set. By Proposition 5.4.13, it follows that the number of resolved openness cases is bounded by the index of $\sim_{\mathcal{A}}$ up to counter limit ℓ. This index is bounded by $(\ell+1)|Q^{\mathcal{A}}|+1$ (see Lemma 5.4.3). As $\ell \leq \zeta$ by Proposition A.8.2, the number of resolved openness cases is in $\mathcal{O}\left(|Q^{\mathcal{A}}|\zeta\right)$ and $\overline{R}$ may increase by at most $(\zeta+1)|Q^{\mathcal{A}}|+1$ words. By adding the potential $\perp$-word of $R \setminus \overline{R}$, we get that R may increase by at most $(\zeta+1)|Q^{\mathcal{A}}|+2$ words.
After that, to handle the counterexample provided by the teacher, R may still increase by at most ζ words (by Lemma 5.4.25).
Let us now study the size of R_f. We denote by $r(i)$ the size of R respectively at the initialization of the learning algorithm when $i = 0$ and after each round i, when $i \geq 1$. We have

$$r(0) = 1,$$
$$\forall i > 0 : r(i) \leq r(i-1) + (\zeta+1)|Q^{\mathcal{A}}| + 2 + \zeta.$$

Recall that the number of rounds is bounded by the total number of **PEQ**,

Proposition 5.4.13. Let $\mathcal{O}_{\leq \ell}$ be an observation table up to $\ell \in \mathbb{N}$. Then, for all $u, v \in \overline{R \cup R\Sigma}$ such that $u \sim_{\mathcal{A}} v$, we have $u \in \mathit{Approx}(v)$. Moreover, all $u \in R \cup R\Sigma \setminus \overline{R \cup R\Sigma}$ belong to the same approximation set.

Lemma 5.4.3. The number of states of $BG_{\leq \ell}(\mathcal{A})$ is at most $(\ell+1) \cdot |Q^{\mathcal{A}}|$.

which is in $\mathcal{O}\left(\zeta^3\right)$ by Proposition A.8.2. Hence, the size of R_f is in $\mathcal{O}\left(|Q^{\mathcal{A}}|\zeta^4\right)$ and the size of $R_f \cup R_f\Sigma$ is in $\mathcal{O}\left(|Q^{\mathcal{A}}||\Sigma|\zeta^4\right)$. □

In the next proposition, we study how the sizes of S and $\widehat{S}$ increase when making an observation table closed, Σ- and $\bot$-consistent. Recall that both S and $\widehat{S}$ do not change when resolving openness cases (see Lemma 5.4.19). Moreover, during the process of making the table consistent (see Lemmas 5.4.20 and 5.4.21), either an approximation set decreases in size or a mismatch is eliminated. Below, we study how many times each of these two cases may happen.

Proposition A.8.4. *Let $\mathcal{O}_{\leq \ell}$ be an observation table and let $\mathcal{O}'_{\leq \ell}$ be the resulting table after making $\mathcal{O}_{\leq \ell}$ closed, Σ- and $\bot$-consistent. Then, with α being the size of $R_f \cup R_f\Sigma$,*

- *$|S'|$ is in $\mathcal{O}\left(\alpha^{4\alpha^2+2}\left|\widehat{S}\right|\right)$ and $\left|\widehat{S}'\right|$ is in $\mathcal{O}\left(\alpha^{4\alpha^2+4}\left|\widehat{S}\right|\right)$.*
- *the number of times that an approximation set decreases is bounded by α^2 and the number of times a mismatch is eliminated is in $\mathcal{O}\left(\alpha^{4\alpha^2+3}\left|\widehat{S}\right|\right)$.*

Proof. Resolving Σ- and $\bot$-inconsistencies during the process of making $\mathcal{O}_{\leq \ell}$ closed and consistent either decreases approximation sets or eliminates mismatches, and the latter case only occurs when resolving a $\bot$-inconsistency (see Lemmas 5.4.20 and 5.4.21).

Let us first study the number of times that an approximation set may decrease. By definition, each processed approximation set is a subset of $R' \cup R'\Sigma$ and there are at most $|R' \cup R'\Sigma|$ such sets. Therefore the number of times an approximation set decreases is bounded by $|R' \cup R'\Sigma|^2$, and thus by

$$\alpha^2. \tag{A.8.i}$$

Let us now study the growth of both S and $\widehat{S}$. By Lemmas 5.4.19 to 5.4.21, S and $\widehat{S}$ may increase when resolving Σ- and $\bot$-inconsistencies only. Let us denote by $\hat{s}$ the initial size of $\widehat{S}$ and by $\hat{s}(i,j)$ the size of the current set $\widehat{S}$ after $i+j$ steps composed of i eliminations of mismatches and j decreases of approximation sets. During such steps, by Lemmas 5.4.20 and 5.4.21, the size of $\widehat{S}$ increases by at most $|R' \cup R'\Sigma| \leq \alpha$ words. Therefore we get:

$$\begin{aligned} \hat{s}(0,0) &= \hat{s}, \\ \forall i+j > 0 : \hat{s}(i,j) &\leq (i+j)\alpha + \hat{s}. \end{aligned} \tag{A.8.ii}$$

Let us introduce another notation: $s(j)$ is the size of the current set S after j decreases of approximation sets and a certain number of eliminations of mismatches such that the last step is one decrease of an approximation set. Thus, $s(0) \leq \hat{s}$, since $S \subseteq \widehat{S}$, and from $s(j-1)$ to $s(j)$, a certain number of mismatches is resolved followed by one decrease of an approximation set. Let us recall that when a mismatch is resolved, the current S is unchanged (see Lemma 5.4.21). It follows that from $s(j-1)$ to $s(j)$, at most:

$$s(j-1)\alpha \text{ mismatches} \tag{A.8.iii}$$

are resolved (indeed for a fixed $s(j-1)$, at most $|R \cup R\Sigma|$ mismatches can be resolved). Hence, from (A.8.iii), we get

$$s(j) \leq \hat{s}(s(j-1)\alpha, j)$$

and, from (A.8.ii), we get

$$s(j) \leq (s(j-1)\alpha + j)\alpha + \hat{s}.$$

It follows that:

$$\begin{aligned} s(0) &\leq \hat{s}, \\ s(j) &\leq s(j-1)\alpha^2 j + \hat{s}. \end{aligned}$$

We thus get

$$s(j) \leq (j+1)(\alpha^2 j)^j \hat{s},$$

via Lemma A.8.1. Recall that there are at most α^2 decreases of approximation sets by (A.8.i). Moreover, to have the table closed, Σ- and $\perp$-consistent, the last such decrease could be followed by several eliminations of mismatches that do not change the size of the current S. Therefore, at the end of the process,

$$|S'| = s(\alpha^2) \leq (\alpha^2 + 1)(\alpha^2\alpha^2)^{\alpha^2} \hat{s}$$

which is in $\mathcal{O}\left(\alpha^{4\alpha^2+2}\left|\hat{S}\right|\right)$. By (A.8.iii), we also get that the total number of eliminations of mismatches is:

$$s(\alpha^2)\alpha \tag{A.8.iv}$$

which is in $\mathcal{O}\left(\alpha^{4\alpha^2+3}\left|\hat{S}\right|\right)$.

Finally we can deduce the size of the resulting set $\hat{S}'$ when the table is closed, Σ- and $\perp$-consistent: it is equal to $\hat{s}(i,j)$ with $i = s(\alpha^2)\alpha$ by (A.8.iv) and $j = \alpha^2$ by (A.8.i). By (A.8.ii), we get

$$\left|\hat{S}'\right| \leq (s(\alpha^2)\alpha + \alpha^2)\alpha + \hat{s}$$

which is in $\mathcal{O}\left(\alpha^{4\alpha^2+4}\left|\hat{S}\right|\right)$. □

As a corollary of Propositions A.8.3 and A.8.4, we obtain a table can be made closed, Σ- and $\perp$-consistent in a finite amount of time. That is, we immediately obtain a proof for Proposition 5.4.18. In view of these two results, we decided to handle a counterexample w to a (partial) equivalence query by adding *Pref*(w) to the current set R. Indeed, an alternative could have been to add *Suff*(w) to $\hat{S}$. It should however be clear that this is not such a good idea because of the exponential growth of $\hat{S}$ as established in Proposition A.8.4. In contrast, the size of R at the end of the learning process is only polynomial (see Proposition A.8.3).

We now study the size of $\hat{S}_f$ which turns out to be exponential.

Proposition A.8.5. *The size of $\hat{S}_f$ is exponential in $\left|Q^{\mathcal{A}}\right|$, $|\Sigma|$ and ζ.*

Proof. We first study the growth of $\hat{S}$ after one round of the learning algorithm. We first make the current table $\mathcal{O}_{\leq \ell}$ closed, Σ- and $\bot$-consistent, and then handle a counterexample to a (partial) equivalence query. For the resulting table $\mathcal{O}'_{\leq \ell'}$, we have by Lemma 5.4.25 and proposition A.8.4 that there is some constant c such that:

$$\left|\hat{S}'\right| \leq c\alpha^{4\alpha^2+4}\left|\hat{S}\right|. \tag{A.8.v}$$

Recall that the number of rounds is bounded by the total number of partial equivalence queries, which is in $\mathcal{O}\left(\zeta^3\right)$ by Proposition A.8.2. Since initially $\hat{S} = \{\varepsilon\}$, we get by (A.8.v) that

$$\left|\hat{S}_f\right| \leq \left(c\alpha^{4\alpha^2+4}\right)^{\mathcal{O}(\zeta^3)}.$$

Since $\alpha = |R_f \cup R_f\Sigma|$ is polynomial in $\left|Q^{\mathcal{A}}\right|$, $|\Sigma|$ and ζ by Proposition A.8.3, it follows that $\left|\hat{S}_f\right|$ is exponential in $\left|Q^{\mathcal{A}}\right|$, $|\Sigma|$ and ζ. □

We get the next corollary about the total number of membership and counter value queries asked by the learner during the execution of the learning algorithm.

Corollary A.8.6. *In the learning algorithm, the number of* **MQ** *and* **CVQ** *is exponential in* $\left|Q^{\mathcal{A}}\right|$, $|\Sigma|$ *and* ζ*.*

Proof. We first study the number of membership and counter value queries after one round of the learning algorithm. Recall that both numbers are polynomial in the size of the current table after resolving an openness, a Σ-inconsistency, a $\bot$-inconsistency, or handling a counterexample (see Lemmas 5.4.19 to 5.4.21 and 5.4.25). Moreover, recall that making the table closed, Σ- and $\bot$-consistent requires

- resolving at most $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|\zeta\right)$ openness cases (by Proposition A.8.3),
- at most α^2 decreases of approximation sets, and
- at most $\mathcal{O}\left(\alpha^{4\alpha^2+3}\left|\hat{S}\right|\right)$ eliminations of mismatches (by Proposition A.8.4),

where $\alpha = |R_f \cup R_f\Sigma| \in \mathcal{O}\left(\left|Q^{\mathcal{A}}\right||\Sigma|\zeta^4\right)$ by Proposition A.8.3. Hence, we get after one round a number of **MQ** and **CVQ** that is exponential in $\left|Q^{\mathcal{A}}\right|$, $|\Sigma|$ and ζ.
Second, as the number of rounds is in $\mathcal{O}\left(\zeta^3\right)$, the total number of **MQ** and **CVQ** during the learning algorithm is again exponential in $\left|Q^{\mathcal{A}}\right|$, $|\Sigma|$ and ζ. □

A.8.3. Proof of Theorem 5.4.1

We are now ready to prove our main theorem establishing the complexity of the learning algorithm (Theorem 5.4.1), which we repeat one more time.

Theorem 5.4.1. *Let $\mathcal{A}$ be the sound ROCA of the teacher and ζ be the length of the longest counterexample returned by the teacher on (partial) equivalence queries. Then,*

- *the L^*_{ROCA} algorithm eventually terminates and returns an ROCA accepting $\mathcal{L}(\mathcal{A})$ and whose size is polynomial in $|Q^{\mathcal{A}}|$ and $|\Sigma|$,*
- *in time and space exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ, and*
- *asking a number of* **PEQ** *in $\mathcal{O}(\zeta^3)$, a number of* **EQ** *in $\mathcal{O}(|Q^{\mathcal{A}}|\zeta^2)$, and a number of* **MQ** *and* **CVQ** *exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ.*

We need a last lemma studying the complexity of the basic operations carried out by the learner during this algorithm. By *basic operations*, given an observation table $\mathcal{O}_{\leq \ell}$, we mean:

- to check whether $u \in \mathit{Approx}(v)$,
- to check whether $u \in \mathit{Pref}(\mathcal{O}_{\leq \ell})$,
- to check whether $\mathcal{O}_{\leq \ell}$ is closed,
- to check whether $\mathcal{O}_{\leq \ell}$ is Σ-consistent,
- to check whether $\mathcal{O}_{\leq \ell}$ is $\perp$-consistent,
- to construct the automaton $\mathcal{H}_{\leq \ell}$ when $\mathcal{O}_{\leq \ell}$ is closed, Σ- and $\perp$-consistent,
- to compute the periodic descriptions $\alpha_1, \dots, \alpha_n$ in $\mathcal{H}_{\leq \ell}$ and construct the ROCAs $\mathcal{A}_{\alpha_1}, \dots, \mathcal{A}_{\alpha_n}$,
- to modify $\mathcal{H}_{\leq \ell}$ to see it as an ROCA.

Lemma A.8.7. *Given an observation table $\mathcal{O}_{\leq \ell}$, each basic operation of the learner is in time polynomial in $|\mathcal{O}_{\leq \ell}|$.*

Proof. Since the actual complexity of the basic operations greatly depends on the implementation details, we give here a very naive complexity. For instance, to check whether $u \in \mathit{Approx}(v)$ with $u, v \in R \cup R\Sigma$, we test for every $s \in S$ if $T(us) = T(vs)$ and $C(us) \neq \perp \wedge C(vs) \neq \perp \Rightarrow C(us) = C(vs)$. This operation is therefore in $\mathcal{O}(|S|)$ which is polynomial in $|\mathcal{O}_{\leq \ell}|$.
Let $u \in \Sigma^*$. Checking whether $u \in \mathit{Pref}(\mathcal{O}_{\leq \ell})$ is equivalent to checking if

$$\exists v \in R \cup R\Sigma, s \in \hat{S} : u \in \mathit{Pref}(vs) \wedge T(vs) = \textbf{yes},$$

leading to a polynomial complexity.
Checking whether the table is closed, *i. e.*,

$$\forall u \in R\Sigma : \mathit{Approx}(u) \cap R \neq \emptyset$$

can be done in time polynomial in $|\mathcal{O}_{\leq \ell}|$. Likewise, checking whether the table is Σ-consistent, *i. e.*,

$$\forall ua \in R\Sigma : ua \in \bigcap_{v \in \mathit{Approx}(u) \cap R} \mathit{Approx}(va),$$

and is $\perp$-consistent, *i. e.*, for all $u, v \in R \cup R\Sigma$ and $s \in S$

$$u \in \mathit{Approx}(v) \Rightarrow C(us) \neq \perp \Leftrightarrow C(vs) \neq \perp.$$

Let n be the index of $\equiv_{\mathcal{O}_{\leq \ell}}$. To construct $\mathcal{H}_{\leq \ell}$, we need to select one repre-

sentative u by equivalence class of $\equiv_{\mathcal{O}_{\leq \ell}}$ and to define the transitions (which implies finding the class of ua). Since n is bounded by $|R|$, the construction of $\mathcal{H}_{\leq \ell}$ has a complexity polynomial in $|\mathcal{O}_{\leq \ell}|$.
By [NL10], we know that finding the periodic descriptions in $\mathcal{H}_{\leq \ell}$ is in polynomial time. The construction of an ROCA from a description is also in polynomial time, by Proposition 5.3.12.
Finally, to see $\mathcal{H}_{\leq \ell}$ as an ROCA, it is sufficient to modify it with a transition function that keeps the counter always equal to 0.
Thus, every basic operation for the learner is in polynomial time in the size of the table. □

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

Proposition 5.3.12. Let $BG(\mathcal{A})$ be the behavior graph of some sound ROCA $\mathcal{A}$ and $\alpha = \tau_0 \dots \tau_{m-1}(\tau_m \dots \tau_{m+k-1})^\omega$ be an ultimately periodic description of $BG(\mathcal{A})$ with offset m and period k. Then, one can construct an ROCA $\mathcal{A}_\alpha$ from α such that

- $\mathcal{L}(\mathcal{A}_\alpha) = \mathcal{L}(\mathcal{A})$, and
- the size of $\mathcal{A}_\alpha$ is polynomial in m, k and *width*$(BG(\mathcal{A}))$.

We finally conclude with the proof of our main theorem.

Proof of Theorem 5.4.1. The number of different kinds of queries is established in Proposition A.8.2 and Corollary A.8.6. By Propositions A.8.3 and A.8.4, the space used by the learning algorithm is mainly the space used to store the observation table which is polynomial (resp. exponential) in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ for $R_f \cup R_f\Sigma$ (resp. for $\hat{S}_f$).
Finally, the algorithm runs in time exponential in $|Q^{\mathcal{A}}|$, $|\Sigma|$ and ζ. Indeed each basic operation of the learner is in time polynomial in the size of the table by Lemma A.8.7, the algorithm is executed in at most $\mathcal{O}(\zeta^3)$ rounds by Proposition A.8.2, and each round results in at most $\mathcal{O}(|Q^{\mathcal{A}}|\zeta)$ openness cases, α^2 decreases of approximation sets, $\mathcal{O}\left(\alpha^{4\alpha^2+3}|\hat{S}|\right)$ eliminations of mismatches, and has to handle one counterexample by Propositions A.8.3 and A.8.4 (where $\alpha = |R_f \cup R_f\Sigma|$). □

Part III.

Validating JSON Documents

JSON Documents and Schemas | 6.

Our focus in the third part *Validating JSON Documents* is to provide a validation algorithm to verify whether a *JSON document* satisfies a set of constraints (given as a *JSON schema*), in a streaming context where the document is received piece by piece. In Section 5.5.3, we presented a potential use case of our learning algorithm for realtime one-counter automata. However, we had to impose constraints on the considered JSON documents. Namely, *objects* which are supposed to be unordered collections of key-value pairs were considered as ordered. In this part, we lift such constraints.

This chapter, based on [BPS23], introduces JSON documents and schemas with more details, alongside the validation problem and the "classical" algorithm, and serves as an introduction for Chapter 7 and its Appendix B which contain our contributions on this subject.

[BPS23]: Bruyère et al. (2023), "Validating Streaming JSON Documents with Learned VPAs"

Chapter contents

6.1. Introduction

JavaScript Object Notation (JSON) has overtaken XML as the de facto standard data-exchange format, in particular for web applications. JSON documents are easier to read for programmers and end users since they only have arrays and objects as structured types. Moreover, in contrast to XML, they do not include named open and end tags for all values, but open and end tags (braces actually) for arrays and objects only. For instance, some components of *Microsoft Azure*, of *GitHub* and of *AWS* can be interacted with via JSON documents. It is also noteworthy that many applications rely on JSON documents to store local configurations, even if they do not have to be shared over a network, *e.g.*, *Visual Studio Code*.

Given the prevalent usage of JSON documents in real-world cases, it is important to verify that a received document is *valid*, in the sense that its structure satisfies some constraints. Naturally, these constraints depend on the exact application. *JSON schema*[1] is a simple schema language that allows users to define them. Many examples can be found on various websites.[2]

Several previous results have been obtained about the formalization of XML schemas and the use of formal methods to validate XML documents (see, *e.g.*, [BCS15; KMV07; Mar+17; NS18; Sch12; SV02]). Recently, a standard to formalize JSON schemas has been proposed and (hand-coded) validation tools

1: `https://json-schema.org/`

2: For instance, `https://www.schemastore.org/json/` lists various schemas, used in real-world cases.

[BCS15]: Boneva et al. (2015), "Schemas for Unordered XML on a DIME"

[KMV07]: Kumar et al. (2007), "Visibly pushdown automata for streaming XML"

[Mar+17]: Martens et al. (2017), "BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema"

[NS18]: Niewerth et al. (2018), "Reasoning About XML Constraints Based on XML-to-Relational Mappings"

[Sch12]: Schwentick (2012), "Foundations of XML Based on Logic and Automata: A Snapshot"

[SV02]: Segoufin et al. (2002), "Validating Streaming XML Documents" for such schemas can be found online[3]. Pezoa et al, in [Pez+16], observe that the standard of JSON documents is still evolving and that the formal semantics of JSON schemas is also still changing. Furthermore, validation tools seem to make different assumptions about both documents and schemas. The authors of [Pez+16] carry out an initial formalization of JSON schemas into formal grammars from which they are able to construct a *batch* validation tool from a given JSON schema.

3: `https://json-schema.org/`

[Pez+16]: Pezoa et al. (2016), "Foundations of JSON Schema"

Whenever a document has to be transmitted over a network, it will be fragmented into multiple smaller chunks of data, which are received piece by piece in the target system. We call this a *streaming* context. This makes the process of checking whether a document is correct with regards to the constraints encoded in a schema harder. The easy solution would be to buffer the document correctly being received in memory but that increases the memory consumption, which may not be appropriate when the validation is done on a server which has to answer quickly and process multiple documents at the same time.

In this chapter, we first properly define JSON documents and their structure. Then, in Section 6.3, we introduce JSON schemas, and, in Section 6.4, the abstractions we consider throughout this part. Finally, Section 6.5 gives an algorithm that validates whether a document satisfies the constraints given as a schema, and its limits. The next chapter presents a different validation algorithm, based on automata learning.

6.2. JSON documents

In this section, we formally define JSON documents [Bra17]. Our presentation is inspired by [Pez+16] and simplifies some aspects for readability. We refer to the official JSON website[4] for a full description. The next section presents a way to describe the structure these documents should follow.

[Bra17]: Bray (2017), "The JavaScript Object Notation (JSON) Data Interchange Format"

[Pez+16]: Pezoa et al. (2016), "Foundations of JSON Schema"

4: `https://www.json.org/json-en.html`

The JSON format defines six different types of *JSON values*:

- `true`, `false` are JSON values.
- `null` is a JSON value.
- Any decimal number (positive, negative) is a JSON value, called a *number*. In particular any number that is an integer is called an *integer*.
- Any finite sequence of Unicode characters starting and ending with `"` is a JSON value, called a *string value*.
- If $v_1, v_2, \dots, v_n$ are JSON values and $k_1, k_2, \dots, k_n$ are *pairwise distinct* string values, then

  $$\{k_1 : v_1, k_2 : v_2, \dots, k_n : v_n\}$$

  is a JSON value, called an *object*. Each $k_i : v_i$ is called a *key-value pair* such that k_i is the *key*. The collection of these pairs is *unordered*. That is, $\{k_1 : v_1, k_2 : v_2\}$ and $\{k_2 : v_2, k_1 : v_1\}$ are considered as being exactly the same object.

$$
\begin{array}{llll}
\textsc{object} & ::= & \{\,\} & \textit{Empty object} \\
 & \mid & \{\textsc{string} : \textsc{value}(, \textsc{string} : \textsc{value})^{*}\} & \textit{Non-empty object} \\
\textsc{array} & ::= & [\,] & \textit{Empty array} \\
 & \mid & [\textsc{value}(, \textsc{value})^{*}] & \textit{Non-empty array} \\
\textsc{value} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \textsc{string} \mid \textsc{number} & \textit{Primitive values} \\
 & \mid & \textsc{object} & \\
 & \mid & \textsc{array} &
\end{array}
$$

Figure 6.1: Formal grammar for JSON documents. For clarity, rules for STRING and NUMBER are not provided here.

```json
{
  "title": "Validating Streaming JSON Documents with Learned VPAs",
  "keywords": ["visibly pushdown automata", "JSON documents", "streaming validation"],
  "conference": {
    "name": "TACAS",
    "year": 2023
  }
}
```

Figure 6.2: A JSON document.

- If $v_1, v_2, \dots, v_n$ are JSON values, then $[v_1, v_2, \dots, v_n]$ is a JSON value, called an *array*. Each v_i is an *element* and the collection thereof is *ordered*. That is, $[v_1, v_2]$ are $[v_2, v_1]$ are two different arrays (if $v_1 \neq v_2$).

JSON values described by the first four items[5] are called *primitive values*. Figure 6.1 gives a formal grammar for JSON values. For clarity, we do not provide rules for string values and numbers, as they follow definitions occurring in many computer languages. In this part, *JSON documents* are supposed to be objects.[6]

5: That is, `true`, `false`, `null`, numbers (including integers), and strings

6: In [Bra17], a JSON document can be any JSON value and duplicated keys are allowed inside objects. In this part, we follow what is commonly used in practice: JSON documents are objects, and keys are pairwise distinct inside objects.

Example 6.2.1. An example of a JSON document is given in Figure 6.2. We can see that this document is an object containing three keys:

- `"title"`, whose associated value is a string value,
- `"keywords"`, whose value is an array containing string values, and
- `"conference"`, whose value is an object. This inner object contains two keys:
  - `"name"`, whose value is a string value, and
  - `"year"`, whose value is an integer.

It is possible to navigate through JSON documents. If J is an object and k is a key, then $J[k]$ is the value v such that the key-value pair $k : v$ appears in J. If J is an array and n is a natural number, then $J[n]$ is the $(n+1)$-th element of J.[7] More generally, values can be retrieved from JSON documents by using *JSON pointers* that are sequences of references as defined previously. For instance, in Example 6.2.1, `J[keywords][1]`, where `J` is the root of the document, allows to retrieve the value `"JSON documents"`.

7: Arrays are indexed from 0.

6.3. JSON schema

As said above, applications that process JSON documents expect them to follow a fixed structure. For instance, one may impose that a document necessarily has keys `"title"` (whose value must be a string) and `"conference"` (whose value must be another object), and, optionally, a key `"keywords"` (whose value is an array of strings). In this part, we focus on *JSON schemas* as tools to define these constraints. We give here a simplified presentation of JSON schemas[8] and refer to the official website[9] for a complete description and to [Pez+16] for a more thorough formalization (*i. e.*, a formal grammar with its syntax and semantics). We say that a JSON document *satisfies* the schema if it verifies the constraints imposed by this schema.

8: In particular, schemas allow one to define constraints subject to the presence of certain values. We do not discuss this here.

9: `https://json-schema.org/`

[Pez+16]: Pezoa et al. (2016), "Foundations of JSON Schema"

A JSON schema is itself a JSON document that uses several keywords that help to shape and restrict the set of JSON documents that this schema specifies, *e. g.*,

- it can be imposed that a string value has a minimum/maximum length or satisfies a pattern expressed by a regular expression;
- that a number belongs to some interval or is a multiple of some number.
- within object schemas, restrictions can be imposed on the key-value pairs of the objects. For example, the value associated with some key has itself to satisfy a certain schema, or some particular keys must be present in the object. A minimum/maximum number of pairs can also be imposed;
- within array schemas, it can be imposed that all elements of the array satisfy a certain schema, or that the array has a minimum/maximum size;
- schemas can be combined with Boolean operations, in the sense that a JSON document must satisfy the conjunction/disjunction of several JSON schemas, or it must not satisfy a certain JSON schema;
- a schema can be the enumeration of certain JSON values;
- a schema can be defined as one referred to by a JSON pointer. This allows a *recursive* structure for the JSON documents satisfying a certain schema.

Example 6.3.1. The schema from Figure 6.3 describes the objects that can have three keys:

- `"title"`, whose associated value must be a string value,
- `"keywords"`, whose value must be an array containing string values, and
- `"conference"`, whose value must be an object.

Among these keys, `"title"` and `"conference"` are required. The JSON document of Figure 6.2 satisfies this JSON schema.

```
{
  "type": "object",
  "required": ["title", "conference"],
  "properties": {
    "title": { "type": "string" },
    "keywords": {
      "type": "array",
      "items": { "type": "string" }
    },
    "conference": {
      "type": "object",
      "required": ["name", "year"],
      "properties": {
        "name": { "type": "string" },
        "year": { "type": "integer" }
      }
    }
  }
}
```

Figure 6.3: A JSON schema.

6.4. Abstract JSON documents and schemas

For the purpose of this part, we consider somewhat *abstract* JSON values, documents, and schemas. The abstractions we introduce here allow us to think of JSON document as words over some alphabet.[10]

10: In turn, this will allow us to actively learn an automaton in Chapter 7.

We do not consider the restrictions that can be imposed on string values and numbers, and we abstract all string values as `s`, and all numbers as `n` (as `i` when they are integers). We denote by

$$\Sigma_{\text{pVal}} = \{\texttt{true}, \texttt{false}, \texttt{null}, \texttt{s}, \texttt{n}, \texttt{i}\}$$

the alphabet composed of the primitive values. We also do not consider *enumerations*.[11]

11: In short, an enumeration specifies a set of allowed values.

Concerning the key-value pairs appearing in objects, each key together with the symbol ":" following the key is abstracted as an alphabet symbol k. We write Σ_{key} for the *finite* alphabet of allowed keys.

We use $\#$ instead of a comma in objects and arrays, and $\prec$ and $\succ$ (resp. $\sqsubset$ and $\sqsupset$) instead of "{" and "}" (resp. "[" and "]"). Finally, we denote by Σ_{JSON} the set of all considered symbols, *i. e.*,

$$\Sigma_{\text{JSON}} = \Sigma_{\text{pVal}} \cup \Sigma_{\text{key}} \cup \{\#, \prec, \sqsubset, \succ, \sqsupset\}.$$

Example 6.4.1. The JSON document given in Example 6.2.1 is abstracted as the word

$$\prec \texttt{title}\ \texttt{s}\ \#\ \texttt{keywords} \sqsubset \texttt{s}\ \#\ \texttt{s}\ \#\ \texttt{s} \sqsupset \# \\ \texttt{conference} \prec \texttt{name}\ \texttt{s} \# \texttt{year}\ \texttt{i} \succ \succ.$$

S	$::=$	v with $v \in \Sigma_{\text{pVal}}$	*Primitive schema*
	$\mid$	$\prec k_1\mathsf{S}_1 \# k_2\mathsf{S}_2 \# \dots \# k_n\mathsf{S}_n \succ$, with $n \geq 0$ and $k_i \neq k_j \in \Sigma_{\text{key}}$ for all $i \neq j \in \{1, \dots, n\}$	*Object schema*
	$\mid$	$\sqsubset \varepsilon \vee \mathsf{S}_1(\#\mathsf{S}_1)^* \sqsupset$	*Array schema (no fixed number)*
	$\mid$	$\sqsubset \mathsf{S}_1 \# \cdots \# \mathsf{S}_1 \sqsupset$, with $n \geq 0$ occurrences of S_1	*Array schema (fixed number)*
	$\mid$	$\mathsf{S}_1 \vee \cdots \vee \mathsf{S}_n$	*Boolean operation (or)*
	$\mid$	$\mathsf{S}_1 \wedge \cdots \wedge \mathsf{S}_n$	*Boolean operation (and)*
	$\mid$	$\neg\mathsf{S}_1$	*Boolean operation (not)*
$\forall j : \mathsf{S}_j$	$::=$	S	*Sub-schemas have the same shape*

Figure 6.4: Extended CFG for an abstracted JSON schema.

We now use the formalism of *extended context-free grammars* (extended CFGs) to define JSON schemas with the abstractions mentioned previously. We recall that in an extended CFG, the right-hand sides of productions are regular expressions over the terminals and non-terminals. Here, we even use *generalized* regular expressions such that intersections and negations are allowed in addition to union, concatenation and Kleene-$*$ operations.

An extended CFG $\mathcal{G}$ defining a JSON schema is given in Figure 6.4. It uses the alphabet Σ_{JSON} of terminals, an alphabet $\mathcal{S} = \{\mathsf{S}, \mathsf{S}_1, \mathsf{S}_2, \dots\}$ of non-terminals (with S being the axiom), and a finite series of productions.[12]

12: In an abuse of notation, the non-terminals of $\mathcal{S}$ are also called schemas.

This extended CFG $\mathcal{G}$ must be *closed*, *i.e.*, whenever it contains a production

$$\mathsf{S} ::= \prec k_1\mathsf{S}_1 \# k_2\mathsf{S}_2 \# \dots \# k_n\mathsf{S}_n \succ,$$

then it also contains *all productions*

$$\mathsf{S} ::= \prec k_{i_1}\mathsf{S}_{i_1} \# k_{i_2}\mathsf{S}_{i_2} \# \dots \# k_{i_n}\mathsf{S}_{i_n} \succ$$

where $(i_1, \dots, i_n)$ is a permutation of $(1, \dots, n)$. Indeed, we recall that objects are unordered collections of key-value pairs.

Example 6.4.2. The JSON schema of Example 6.3.1 can be defined as follows:

$$\mathsf{S}_0 ::= \prec \texttt{title}\ \mathsf{S}_1 \# \texttt{keywords}\ \mathsf{S}_2 \# \texttt{conference}\ \mathsf{S}_3 \succ$$
$$\mathsf{S}_0 ::= \prec \texttt{title}\ \mathsf{S}_1 \# \texttt{conference}\ \mathsf{S}_3 \succ$$
$$\mathsf{S}_1 ::= \texttt{s}$$
$$\mathsf{S}_2 ::= \sqsubset \varepsilon \vee \mathsf{S}_1(\#\mathsf{S}_1)^* \sqsupset$$
$$\mathsf{S}_3 ::= \prec \texttt{name}\ \mathsf{S}_1 \# \texttt{year}\ \mathsf{S}_4 \succ$$
$$\mathsf{S}_4 ::= \texttt{i}$$

where we add to the first, second, and fifth productions all the related productions with key permutations. The axiom of this grammar is S_0.

6.4.1. Semantics

Let us now describe the *semantics* of a closed extended CFG $\mathcal{G}$ defining a JSON schema. Let $\mathsf{S} \in \mathcal{S}$ be a non-terminal symbol and J be a valid JSON value.[13]

13: That is, J is either a primitive value or a well-formed object or array.

We say that J *satisfies* S, or J is *valid* with regards to S, denoted by $J \vDash \mathrm{S}$, if one of the following holds:

- S is a primitive schema v and $J = v$.
- S is an object schema $\prec k_1\mathrm{S}_1 \# k_2\mathrm{S}_2 \# \dots \# k_n\mathrm{S}_n \succ$, J is an object $\prec k_1 v_1 \# k_2 v_2 \# \dots \# k_n v_n \succ$ such that $v_i \vDash \mathrm{S}_i$ for every $i \in \{1, \dots, n\}$.
- S is an array schema $\sqsubset \varepsilon \vee \mathrm{S}_1(\#\mathrm{S}_1)^* \sqsupset$, J is an array, and for each element v of J, we have $v \vDash \mathrm{S}_1$.
- S is an array schema $\sqsubset \mathrm{S}_1 \# \dots \# \mathrm{S}_1 \sqsupset$ with $n \geq 0$ occurrences of S_1, J is an array of size n, and for each element v of J, we have $v \vDash \mathrm{S}_1$.
- S is $\mathrm{S}_1 \vee \mathrm{S}_2 \vee \dots \vee \mathrm{S}_n$ and there exists $i \in \{1, \dots, n\}$ such that $J \vDash \mathrm{S}_i$.
- S is $\mathrm{S}_1 \wedge \mathrm{S}_2 \wedge \dots \wedge \mathrm{S}_n$ and for all $i \in \{1, \dots, n\}$ we have $J \vDash \mathrm{S}_i$.
- S is $\neg\mathrm{S}_1$ and $J \nvDash \mathrm{S}_1$.

Let us make some comment about this semantics. If J is an object satisfying S with respect to the production $\mathrm{S} ::= \prec k_1\mathrm{S}_1 \# k_2\mathrm{S}_2 \# \dots \# k_n\mathrm{S}_n \succ$, as $\mathcal{G}$ is closed, then the document J with any permutation of its key-value pairs also satisfies S as $\mathcal{G}$ contains all the related productions with key permutations. Notice also that an array J can be composed of any number of elements or of a fixed number of elements, all satisfying the same schema. Moreover, the empty object $\prec\succ$ and the empty array $\sqsubset\sqsupset$ are allowed.

We denote by $\mathcal{L}(\mathcal{G})$ the set of all JSON values J satisfying the axiom S of $\mathcal{G}$. Hence, when a JSON schema, given as a grammar $\mathcal{G}$, defines a set of JSON documents, this set of documents is the language $\mathcal{L}(\mathcal{G})$.[14]

14: In the sequel, we abuse notation and speak of JSON values, documents and schemas instead of abstractions.

Example 6.4.3. Consider the closed extended CFG $\mathcal{G}$ of Example 6.4.2. The JSON document J of Example 6.4.1, equal to

$$\prec \mathtt{title}\ \mathtt{s} \# \mathtt{keywords} \sqsubset \mathtt{s} \# \mathtt{s} \# \mathtt{s} \sqsupset \# \\ \mathtt{conference} \prec \mathtt{name}\ \mathtt{s} \# \mathtt{year}\ \mathtt{i} \succ \succ.$$

satisfies the axiom S_0 of $\mathcal{G}$. This is also the case for the document J' equal to

$$\prec \mathtt{conference} \prec \mathtt{name}\ \mathtt{s} \# \mathtt{year}\ \mathtt{i} \succ \# \mathtt{title}\ \mathtt{s} \# \\ \mathtt{keywords} \sqsubset \mathtt{s} \# \mathtt{s} \# \mathtt{s} \sqsupset \succ.$$

Therefore, J, J' are both valid and $J, J' \in \mathcal{L}(\mathcal{G})$.

The set of all JSON values can be defined by a particular grammar as given in the next lemma.

Lemma 6.4.4. *For a given set of keys Σ_{key}, the set of all valid JSON values (i.e., primitive values or well-formed objects and arrays) J is equal to $\mathcal{L}(\mathcal{G}_{\mathrm{U}})$ where $\mathcal{G}_{\mathrm{U}}$ is the closed extended CFG given in Figure 6.5, called* universal, *with U being the axiom.*

Remark 6.4.5. As mentioned in [Pez+16], we suppose to work with *well-formed* extended CFGs that avoid problematic situations like in the production $\mathrm{S} ::= \neg\mathrm{S}$ or the productions $\mathrm{S} ::= \mathrm{S}_1 \vee \mathrm{S}_2$, $\mathrm{S}_2 ::= \mathrm{S}$. That is, we

[Pez+16]: Pezoa et al. (2016), "Foundations of JSON Schema"

$$\begin{array}{lcll} \mathrm{U} & ::= & v \text{ for all } v \in \Sigma_{\text{pVal}} & \textit{Primitive values} \\ & \mid & \sqsubset \varepsilon \lor U(\#U)^{*} \sqsupset & \textit{Arrays} \\ & \mid & \prec k_1 U \# \dots \# k_n U \text{ for all sequences } (k_1, \dots, k_n), n \geq 0 \text{, of pairwise} & \textit{Objects} \\ & & \text{distinct keys of } \Sigma_{\text{key}} & \end{array}$$

Figure 6.5: Universal extended CFG.

avoid grammars with cyclic definitions inside the productions that involve Boolean operations (see [Pez+16] for more details).

6.5. Validation of JSON documents

Let us now explain the *classical* algorithm used in many implementations for validating a JSON document J_0 against a JSON schema S_0. It is a recursive algorithm that follows the semantics of a closed extended CFG $\mathcal{G}$ defining this schema. For instance, if the current value J is an object, we iterate over each key-value pair in J and its corresponding sub-schema in the current schema S. Then, J satisfies S if and only if the values in the key-value pairs all satisfy their corresponding sub-schema.

Algorithm 6.1 gives a pseudo-code for the classical validation algorithm. We refer to [Pez+16] for a complexity analysis and a more thorough discussion.

[Pez+16]: Pezoa et al. (2016), "Foundations of JSON Schema"

As long as the grammar $\mathcal{G}$ does not contain any Boolean operations, this algorithm is straightforward and linear in the size of both the initial document J_0 and schema S_0. However, if $\mathcal{G}$ contains Boolean operations, then the current value J may be processed multiple times. For instance, to verify whether J satisfies $S_1 \land S_2 \land \dots \land S_n$, J must be validated against each S_i.

Hence, when we want to validate a document in a *streaming* context,[15] the classical algorithm has to buffer the document being received in memory, which increases the amount of memory required for an execution. This may be problematic when the validation process runs on a server that has to answer quickly, especially when multiple processes run in parallel. In the next chapter, we present a different approach that requires less memory.

15: That is, when we do not know the whole document beforehand but we receive it piece by piece.

Algorithm 6.1: Classical validation algorithm for JSON documents.

Require: A JSON value J and a JSON schema given as a non-terminal symbol S in the closed extended CFG
Ensure: Returns **yes** if and only if $J \vDash \mathrm{S}$

```
function ClassicalValidation($J$, S)
    if S is a primitive schema then return $J = \mathrm{S}$
    else if S is an object schema then
        if $J$ is not an object then return no
        if the same key $k$ appears multiple times in $j$ then return no
        if the set of keys appearing in $J$ differs from the set of keys of S then return no
        for all key $k$ appearing in $J$ do
            Let $\mathrm{S}_k$ be the non-terminal following $k$ in the production of S
            if ClassicalValidation($J[k], \mathrm{S}_k$) = no then
                return no
    else if S is an array schema then
        if $J$ is not an array then return no
        Let $n$ be the length of $J$
        if S requires $m$ values and $n \neq m$ then return no
        Let $\mathrm{S}'$ be the non-terminal appearing in the production of S
        for all $i \in \{0, \dots, n\}$ do
            if ClassicalValidation($J[i], \mathrm{S}'$) = no then
                return no
    else if $\mathrm{S} = \mathrm{S}_1 \vee \cdots \vee \mathrm{S}_n$ then
        if for all $i \in \{1, \dots, n\}$, ClassicalValidation($J, \mathrm{S}_i$) = no then
            return no
    else if $\mathrm{S} = \mathrm{S}_1 \wedge \cdots \wedge \mathrm{S}_n$ then
        if there exists $i \in \{1, \dots, n\}$ such that ClassicalValidation($J, \mathrm{S}_i$) = no then
            return no
    else if $\mathrm{S} = \neg \mathrm{S}'$ then
        if ClassicalValidation($J, \mathrm{S}_i$) = yes then return no
    return yes
```

Validating JSON Documents by Learning Automata

7.

In this chapter, based on [BPS23], we present a validation algorithm for JSON documents against a JSON schema that relies on learning an automaton accepting the same set of documents as the schema. More precisely, we learn a special kind of pushdown automaton, called *visibly pushdown automata*, thanks to a learning algorithm called $\mathtt{TTT}_{\mathrm{VPL}}$ [Isb15].

[BPS23]: Bruyère et al. (2023), "Validating Streaming JSON Documents with Learned VPAs"

[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

Given the exponential number of documents due to the exponential number of permutations of key-value pairs inside an object (see Chapter 6), we fix an order over the keys and learn an automaton that only accepts documents following this order. We then construct a tool telling us how to jump around in the automaton in order to also accept documents that do not follow this order. We also present and discuss experimental results obtained by validating documents against real-world JSON schemas. Technical proofs and details are deferred to Appendix B.

Chapter contents

7.1. Introduction

In this chapter, we again consider the problem of validating a JSON document against a JSON schema. We assume that the schema is given as an extended context-free grammar, as was presented in Section 6.4. That is, we rely on the formalization work of [Pez+16].

[Pez+16]: Pezoa et al. (2016), "Foundations of JSON Schema"

We propose here a *streaming* validation algorithm, *i.e.*, an algorithm that works on documents that are received piece by piece (such as what happens for a web server). To our knowledge, this is the first JSON validation algorithm that is streaming. For XML, works that study streaming document validation base such algorithms on the construction of some automaton (see, *e.g.*, [SV02], for XML). In the previous part, we first experimented with one-counter automata for this purpose. We submit that *visibly-pushdown automata* (VPAs) are a better fit for this task — this is in line with [KMV07], where the same was proposed for streaming XML documents. In contrast to one-counter automata,[1] we show that VPAs are expressive enough to capture the language of JSON documents satisfying any JSON schema.

[SV02]: Segoufin et al. (2002), "Validating Streaming XML Documents"

[KMV07]: Kumar et al. (2007), "Visibly pushdown automata for streaming XML"

1: By nesting objects and arrays, we obtain a set of JSON documents encoding the language composed of all words $a^n b^m c^m d^n$ (with $m, n \in \mathbb{N}$), a context-free language that requires two counters.

We also explain that *active learning à la* Angluin [Ang87] (see Section 3.2) is a good alternative to the automatic construction of such a VPA from the formal semantics of a given JSON schema. This is possible in the presence of labeled examples or a computer program that can answer membership and (approximate) equivalence queries about a set of JSON documents. This learning approach has two advantages. First, we derive from the learned VPA a streaming validator for JSON documents. Second, by automatically learning an automaton representation, we circumvent the need to write a schema and subsequently validate that it represents the desired set of JSON documents. Indeed, it is well known that one of the highest bars that users have to clear to make use of formal methods is the effort required to write a formal specification, in this case, a JSON schema.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

This chapter is structured as follows. In Section 7.2, we recall the concept of VPA and the structure we impose for our learning context. Namely, we work on *1-single entry VPAs* [Alu+05; Isb15] where all call transitions lead to the initial state, and that admit a unique minimal automaton. We also recall how to learn such an automaton using TTT [Isb15]. Then, in Section 7.3, we argue that there always exists such a VPA that accepts the same set of JSON documents as an (abstract) JSON schema, implying that our learning approach is feasible and makes sense. Then, in Section 7.4, we present our new streaming validation algorithm based on learned VPAs. This algorithm relies on a specific graph that abstracts the transitions of the VPA with respect to the objects that appear in the accepted JSON documents. We define this graph and prove several useful properties before providing the validation algorithm, for which we prove time and space complexity results and its correctness. Finally, in Section 7.5 we discuss the implementation of our algorithms and show experimental results comparing our validation approach with the classical algorithm (see Section 6.5). Technical proofs and details are deferred to Appendix B.

[Alu+05]: Alur et al. (2005), "Congruences for Visibly Pushdown Languages"
[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

7.2. Visibly pushdown automata

In Definition 4.3.1, we introduced the notion of *pushdown alphabet*. For simplicity, we restate the definition here and slightly adjust it for the sake of this chapter.

Definition 7.2.1 (Pushdown alphabet)**.** A *pushdown alphabet*, denoted by $\widetilde{\Sigma} = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$, is the union of three disjoint alphabets:

- Σ_c is the set of *calls* where every call pushes a symbol to the stack,
- Σ_r is the set of *returns* where every return pops a symbol from the

stack, and
- Σ_{int} is the set of *internal symbols* where an internal symbol does not change the stack.

In this part, we work with the particular alphabet of return symbols

$$\Sigma_r = \{\bar{a} \mid a \in \Sigma_c\}.$$

Let us now introduce tools that will be used throughout the chapter to guarantee that a JSON value is well-formed, starting with a function giving us the number of unmatched call and return symbols.[2]

2: This is similar to the notion of counter value, introduced in Definition 4.3.2.

Definition 7.2.2 (Balance). The *call/return balance* function $\beta : \Sigma^* \to \mathbb{Z}$ is defined as

$$\beta(ua) = \beta(u) + \begin{cases} 1 & \text{if } a \in \Sigma_c, \\ -1 & \text{if } a \in \Sigma_r, \\ 0 & \text{if } a \in \Sigma_{int}. \end{cases}$$

We can then define *well-matched* words as the set of words for which the balance is null. That is, we see the same number of return and call symbols and, in any prefix of the words, the number of return symbols never exceeds the number of call symbols. The following definition gives a more constructive approach (*i. e.*, it explains how to obtain a well-matched word).

Definition 7.2.3 (Well-matched words). Given a pushdown alphabet $\widetilde{\Sigma}$, the set $\mathrm{WM}(\widetilde{\Sigma})$ of *well-matched* words over $\widetilde{\Sigma}$ is defined inductively:

- $\varepsilon \in \mathrm{WM}(\widetilde{\Sigma})$,
- $\forall a \in \Sigma_{int}, a \in \mathrm{WM}(\widetilde{\Sigma})$,
- if $w, w' \in \mathrm{WM}(\widetilde{\Sigma})$, then $ww' \in \mathrm{WM}(\widetilde{\Sigma})$,
- if $a \in \Sigma_c, w \in \mathrm{WM}(\widetilde{\Sigma})$, then $aw\bar{a} \in \mathrm{WM}(\widetilde{\Sigma})$.

The *depth* of a well-matched word w, denoted by $\mathit{depth}(w)$, is the maximum number of unmatched call symbols among the prefixes of w, *i. e.*,

$$\mathit{depth}(w) = \max_{u \in \mathit{Pref}(w)} \beta(u).$$

Observe that, for all $w \in \mathrm{WM}(\widetilde{\Sigma})$, we have $\beta(u) \geq 0$ for each prefix u of w and $\beta(u) \leq 0$ for each suffix u of w.

We now give the definition of VPAs [AM04]. In short, it is a *nondeterministic* finite automaton augmented with a stack. Each time we read a call, we *push* a symbol (given by the transition) on the stack. Conversely, a return transition can only be triggered when the symbols at the top of the stack and given by the transition are the same, in which the top symbol of the stack is *popped*.

[AM04]: Alur et al. (2004), "Visibly pushdown languages"

Definition 7.2.4 (Visibly pushdown automaton). A *visibly pushdown automaton* (*VPA*, for short) is a tuple $\mathcal{A} = (\widetilde{\Sigma}, \Gamma, Q, q_0, F, \delta)$ where

- $\widetilde{\Sigma}$ is a pushdown alphabet,
- Γ is the *stack alphabet*,
- Q is the finite non-empty set of states, with $q_0 \in Q$ the initial state,

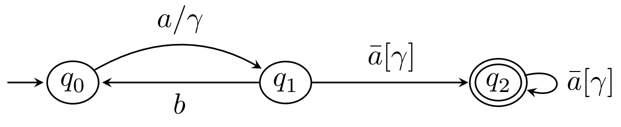


Figure 7.1: A sample VPA.

- ▶ $F \subseteq Q$ is the finite non-empty set of final states, and
- ▶ δ is a finite set of *transitions* of the form $\delta = \delta_c \cup \delta_r \cup \delta_{int}$ where
  - $\delta_c \subseteq (Q \times \Sigma_c) \times (Q \times \Gamma)$ is the set of *call* transitions. We write $\delta_c(q, a)$ to denote the set of pairs (p, γ) such that $((q, a), (p, \gamma)) \in \delta$ and $q \xrightarrow{a/\gamma} p$ when $(p, \gamma) \in \delta_c(q, a)$.
  - $\delta_r \subseteq (Q \times \Sigma_r \times \Gamma) \times Q$ is the set of *return* transitions. We write $\delta_r(q, a, \gamma)$ to denote the set of states p such that $((q, a, \gamma), p) \in \delta$ and $q \xrightarrow{a[\gamma]} p$ when $p \in \delta_r(q, a, \gamma)$.
  - $\delta_{int} \subseteq (Q \times \Sigma_{int}) \times Q$ is the set of *internal* transitions. We write $\delta_r(q, a)$ to denote the set of states p such that $((q, a), p) \in \delta$ and $q \xrightarrow{a} p$ when $p \in \delta_{int}(q, a)$.

As for DFAs, VOCAs, and so on, we add a superscript to indicate which automaton is considered, and missing symbols are quantified existentially. Unlike for NFAs, ROCAs, and so on, we refrain from defining *runs* due to the three different transition functions.

Example 7.2.5. Let $\widetilde{\Sigma} = (\{a\}, \{\bar{a}\}, \{b\})$ be a pushdown alphabet and $\Gamma = \{\gamma\}$ be a stack alphabet. A 3-state VPA $\mathcal{A}$ is given in Figure 7.1.
Call transitions give the input symbol and the stack symbol to push, separated by a slash, *e.g.*, $q_0 \xrightarrow{a/\gamma} q_1$. Likewise, return transition give the input symbol, followed by the stack symbol between square brackets, *e.g.*, $q_0 \xrightarrow{\bar{a}[\gamma]} q_2$. Finally, internal transitions simply give their input symbol.

7.2.1. Semantics

Let us now describe the semantics of a VPA. Recall that for VOCAs and ROCAs, we had to keep track of the counter value when defining counted runs (see Definition 5.2.3). We define *stacked runs* similarly, by keeping track of the current stack contents. That is, we consider *configurations* which are pairs (q, σ) where $q \in Q$ is a state and $\sigma \in \Gamma^*$ is the stack content. We define the transitions between configurations $(q, \sigma), (q', \sigma')$ as follows:

- ▶ $(q, \sigma) \xrightarrow{a} (q', \sigma)$, with $a \in \Sigma_{int}$.
- ▶ $(q, \sigma) \xrightarrow{a/\gamma} (q', \gamma \cdot \sigma)$, with $a \in \Sigma_c$ and $\gamma \in \Gamma$.
- ▶ $(q, \gamma \cdot \sigma) \xrightarrow{a[\gamma]} (q', \sigma)$, with $a \in \Sigma_r$ and $\gamma \in \Gamma$.

Again, missing symbols in a transition are quantified existentially. Observe that stack symbols are pushed to the left of the stack.

Definition 7.2.6 (Stacked runs)**.** Let $\mathcal{A}$ be a VPA. A *stacked run* of $\mathcal{A}$ is either a configuration (p_0, σ_0) or a nonempty sequence of transitions

$$(p_0, \sigma_0) \xrightarrow{a_1} (p_1, \sigma_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (p_n, \sigma_n).$$

We denote by *sruns*$(\mathcal{A})$ the set of all counted runs of $\mathcal{A}$.

This allows us to easily define when a word is accepted: we start from the initial configuration (q_0, ε), read a word w, and must end in a configuration (p, ε) with $p \in F$. We highlight that we request that the stack is empty for a word to be accepted.

Definition 7.2.7 (Visibly pushdown language)**.** The language accepted by a VPA $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{w \in \widetilde{\Sigma}^* \mid \exists q \in F : (q_0, \varepsilon) \xrightarrow{w} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})\}.$$

A language L is called a *visibly pushdown language* (*VPL*, for short) if there is a VPA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

Notice that a VPL is only composed of well-matched words. This differs from the original definition of VPAs in [Alu+05] where ill-matched words can also be accepted.

[Alu+05]: Alur et al. (2005), "Congruences for Visibly Pushdown Languages"

Let us now define a relation over states that tells us whether it is possible to reach the configuration (p, ε) from the configuration (q, ε). That is, there must exist a word w such that $\beta(w) = 0$ (*i.e.*, a well-matched word) and $(p, \varepsilon) \xrightarrow{w} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})$.

Definition 7.2.8 (Reachability relation)**.** Given a VPA $\mathcal{A}$ over $\widetilde{\Sigma}$, the *reachability relation* $\mathrm{Reach}_{\mathcal{A}}$ of $\mathcal{A}$ is:

$$\mathrm{Reach}_{\mathcal{A}} = \{(q, q') \in Q \times Q \mid \exists w \in \mathrm{WM}(\widetilde{\Sigma}) : (q, \varepsilon) \xrightarrow{w} (q', \varepsilon)\}.$$

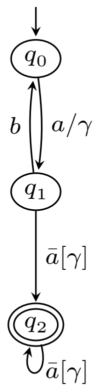
Finally, we say that $p \in Q$ is a *bin state* if there exists no stacked run of the form $(q, \varepsilon) \xrightarrow{w} (p, \sigma) \xrightarrow{w'} (q', \varepsilon)$ with $q \in Q_0$ and $q' \in F$. If a VPA $\mathcal{A}$ has bin states, those states can be removed from Q as well as the transitions containing bin states without modifying the accepted language.

Example 7.2.9. Let us continue Example 7.2.5, *i.e.*, let $\mathcal{A}$ be the VPA of Figure 7.1, which is repeated in the margin.
The word $aba\bar{a}\bar{a}$ is accepted by $\mathcal{A}$. Indeed, we have the following stacked run:

$$\begin{aligned}(q_0, \varepsilon) &\xrightarrow{a} (q_1, \gamma) \xrightarrow{b} (q_0, \gamma) \xrightarrow{a} (q_1, \gamma^2) \\ &\xrightarrow{\bar{a}} (q_2, \gamma) \xrightarrow{\bar{a}} (q_2, \varepsilon) \in \mathit{sruns}(\mathcal{A}).\end{aligned}$$

As $q_2 \in F$ and the stack is empty, we conclude that $w \in \mathcal{L}(\mathcal{A})$. One can show that

$$\mathcal{L}(\mathcal{A}) = \{a(ba)^n \bar{a}^{n+1} \mid n \in \mathbb{N}\}.$$



7.2.2. Minimal deterministic visibly pushdown automata and properties

Given a VPA $\mathcal{A}$, we say that it is *deterministic* if $\mathcal{A}$ does not have two distinct transitions with the same left-hand side. By *left-hand side*, we mean (q, a) for a call transition $(q, a, q', \gamma) \in \delta_c$ or an internal transition $(q, a, q') \in \delta_{int}$, and (q, a, γ) for a return transition $(q, a, \gamma, q') \in \delta_r$. The following theorem states that a VPA can always be made deterministic [Alu+05; Tan09]. We briefly describe the construction, as our validation algorithm will use similar tricks.

[Alu+05]: Alur et al. (2005), "Congruences for Visibly Pushdown Languages"
[Tan09]: Tang (2009), "A Tighter Bound for the Determinization of Visibly Pushdown Automata"

Theorem 7.2.10. *For any VPA $\mathcal{A}$ over $\widetilde{\Sigma}$, one can construct a deterministic VPA $\mathcal{B}$ over $\widetilde{\Sigma}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. Moreover, the size of $\mathcal{B}$ is in $\mathcal{O}\left(2^{|\mathcal{A}|^2}\right)$ and the size of its stack alphabet is in $\mathcal{O}\left(|\Sigma_c| \cdot 2^{|\mathcal{A}|^2}\right)$.*

Proof. Let $\mathcal{A}$ be a VPA over $\widetilde{\Sigma}$. The states of $\mathcal{B}$ are subsets R of the reachability relation $\mathrm{Reach}_{\mathcal{A}}$ of $\mathcal{A}$ and the stack symbols of $\mathcal{B}$ are of the form (R, a) with $R \subseteq \mathrm{Reach}_{\mathcal{A}}$ and $a \in \Sigma_c$.
Let $w = u_1 a_1 u_2 a_2 \dots u_n a_n u_{n+1}$ be such that $n \geq 0$ and $u_i \in \mathrm{WM}(\widetilde{\Sigma})$, $a_i \in \Sigma_c$ for all i. That is, we decompose w in terms of its unmatched call symbols. For all i, let R_i be equal to

$$\{(p, q) \mid (p, \varepsilon) \xrightarrow{u_i} (q, \varepsilon)\}.$$

Then, after reading w, the deterministic VPA $\mathcal{B}$ has its current state equal to R_{n+1} and its stack containing $(R_n, a_n) \dots (R_2, a_2)(R_1, a_1)$. Assume we are reading the symbol a after w, then $\mathcal{B}$ performs the following transition from R_{n+1}:

- if $a \in \Sigma_c$, then push (R_{n+1}, a) on the stack and go to the state $R = \mathbb{I}_Q$ (a new unmatched call symbol is read),
- if $a \in \Sigma_{int}$, then go to the state

  $$R = \{(p, q) \mid \exists (p, p') \in R_{n+1} : (p', a, q) \in \delta_{int}\}$$

  (u_{n+1} is extended to the well-matched word $u_{n+1}a$),
- if $a \in \Sigma_r$, then pop (R_n, a_n) from the stack if $\bar{a}_n = a$, and go to the state

  $$R = \{(p, q) \mid \exists (p, p') \in R_n, (p', a_n, r', \gamma) \in \delta_c, \\ (r', r) \in R_{n+1}, (r, a, \gamma, q) \in \delta_r\}$$

  (the call symbol a_n is matched with the return symbol $a = \bar{a}_n$, leading to the well-matched word $u_n a_n u_{n+1} a$).

Finally the initial state of $\mathcal{B}$ is $\mathbb{I}_{Q_0}$ and its final states are sets R containing some (p, q) with $p \in Q_0$ and $q \in F$. □

The family of VPLs is closed under several natural operations, as given in the next theorem [AM04].

[AM04]: Alur et al. (2004), "Visibly pushdown languages"

Theorem 7.2.11. *Let L_1 and L_2 be two VPLs over $\widetilde{\Sigma}$. Then, $L_1 \cup L_2$, $L_1 \cap L_2$, $\mathrm{WM}(\widetilde{\Sigma}) \setminus L_1$, $L_1 \cdot L_2$, and L_1^* are VPLs over $\widetilde{\Sigma}$.*

Though a VPL L in general does not have a unique minimal deterministic VPA $\mathcal{A}$ accepting L, imposing the following subclass leads to a unique minimal acceptor, called *1-single entry visibly pushdown automata* [Alu+05; Isb15].[3]

[Alu+05]: Alur et al. (2005), "Congruences for Visibly Pushdown Languages"
[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

3: The definitions provided by Alur *et al.* and by Isberner differ slightly. We follow the one in [Isb15].

Definition 7.2.12 (1-module single entry visibly pushdown automata)**.** A *1-module single entry visibly pushdown automata* (*1-SEVPA*, for short) is a deterministic VPA $\mathcal{A}$ such that

- ▶ its stack alphabet Γ is equal to $Q \times \Sigma_c$, and
- ▶ all its call transitions $(q, a, q', \gamma) \in \delta_c$ are such that $q' = q_0$ and $\gamma = (q, a)$.

Theorem 7.2.13 ([Alu+05])**.** *For any VPL L, there exists a unique minimal (with regards to the number of states) 1-SEVPA accepting L, up to a renaming of the states.*[4]

4: This 1-SEVPA may be exponentially bigger than the size of a VPA accepting L.

Let us remark two facts about minimal 1-SEVPAs. First, given a minimal 1-SEVPA $\mathcal{A}$, there may exist a smaller VPA accepting the same language (that is therefore not a 1-SEVPA). Second, the transition relation of $\mathcal{A}$ is a total function, meaning that $\mathcal{A}$ may have a bin state (that is unique, in case of existence).

7.2.3. Learning visibly pushdown automata

In Section 3.2, we explained how one can apply the L^* algorithm to actively learn a DFA [Ang87]. This algorithm necessitates membership and equivalence queries. Isberner *et al.* [IHS14b] proposed a variation of L^*, called the TTT algorithm, which improves the efficiency of the L^* algorithm by eliminating redundancies in counterexamples provided by the teacher.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

[IHS14b]: Isberner et al. (2014), "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning"

In [Isb15], an efficient learning algorithm for VPLs is given by extending Angluin's learning algorithm, but using exactly the same types of queries. The Myhill-Nerode congruence for regular languages is extended to VPLs as follows [Alu+05; Isb15].

Definition 7.2.14 (Extended Myhill-Nerode congruence)**.** Given a pushdown alphabet $\widetilde{\Sigma}$ and a VPL L over $\widetilde{\Sigma}$, we define the set $\mathrm{CP}(\widetilde{\Sigma})$ of *context pairs*:[6]

$$\mathrm{CP}(\widetilde{\Sigma}) = \{(u, v) \in (\mathrm{WM}(\widetilde{\Sigma}) \cdot \Sigma_c)^* \times \mathit{Suff}(\mathrm{WM}(\widetilde{\Sigma})) \mid \beta(u) = -\beta(v)\}.$$

Then, the *Myhill-Nerode congruence for VPL* is the relation $\simeq_L \subseteq \mathrm{WM}(\widetilde{\Sigma}) \times \mathrm{WM}(\widetilde{\Sigma})$ such that $w \simeq_L w'$ if and only if

$$\forall (u, v) \in \mathrm{CP}(\widetilde{\Sigma}) : uwv \in L \Leftrightarrow uw'v \in L.$$

6: Notice that a non-empty word in $(\mathrm{WM}(\widetilde{\Sigma}) \cdot \Sigma_c)^*$ is an element of $\mathit{Pref}(\mathrm{WM}(\widetilde{\Sigma}))$ ending with a call symbol.

The minimal 1-SEVPA accepting L, described in Theorem 7.2.13, is constructed from $\simeq_L$ such that its states are the equivalence classes of $\simeq_L$.

Finally, we state that one can actively learn a minimal 1-SEVPA accepting a given VPL [Isb15].

Theorem 7.2.15. *Let L be a VPL over $\widetilde{\Sigma}$, n be the index of $\simeq_L$, and ζ be the length of the longest counterexample returned by the teacher for an* **EQ**. *Then, one can learn the minimal 1-SEVPA accepting L with*

- *at most $n - 1$ equivalence queries and*
- *a number of membership queries polynomial in n, $|\Sigma|$, and $\log \zeta$.*

In his PhD Thesis [Isb15], Isberner designed an adaptation of `TTT` to VPLs, which we denote $\texttt{TTT}_{\text{VPL}}$.

7.3. Visibly pushdown automata for abstracted JSON schemas

In Sections 6.2 and 6.3, we introduced JSON documents, which can be used to store and transfer information, as well as JSON schemas, which can be used to describe constraints a document should satisfy. We say that a JSON value is valid for a schema, if it satisfies the semantics of the schema. Furthermore, we abstracted documents and schemas in Section 6.4 and provided a formal grammar that is equivalent to a schema. In this section, we claim that there always exists a VPA for that formal grammar. That is, given a schema, one can construct a VPA accepting exactly the set of documents that are valid.

First of all, we have to give the pushdown alphabet corresponding to the symbols of the JSON schema. As in Section 6.4, we assume we have two alphabets. The first, Σ_{pVal}, contains all (abstracted) primitive values, *i. e.*,

$$\Sigma_{\text{pVal}} = \{\texttt{true}, \texttt{false}, \texttt{null}, \texttt{s}, \texttt{n}, \texttt{i}\},$$

while the second, Σ_{key}, is composed of the keys that are defined in the JSON schema. Recall that the colon symbol following a key in an object is part of the abstract key symbol. That is, `"key":` is treated as a single symbol in the key alphabet. Finally, commas are replaced by the $\#$ symbol, while { (resp. }, [, and]) are written $\prec$ (resp. $\succ$, $\sqsubset$, and $\sqsupset$).

We then define a pushdown alphabet $\widetilde{\Sigma}_{\text{JSON}}$ where

- the set of call symbols is $\{\prec, \sqsubset\}$ (*i. e.*, the symbols opening an object or an array),
- the set of return symbols is $\{\succ, \sqsupset\}$ (*i. e.*, the symbols closing an object or an array), and
- the set of internal symbols is $\Sigma_{\text{pVal}} \cup \Sigma_{\text{key}} \cup \{\#\}$.

Then, any (abstract) JSON document must be a well-matched word over $\widetilde{\Sigma}_{\text{JSON}}$ (*i. e.*, in $\text{WM}(\widetilde{\Sigma}_{\text{JSON}})$) to be valid.

Finally, for a closed extended CFG $\mathcal{G}$ defining a JSON schema, let us write $\mathcal{L}(\mathcal{G})$ for the set of JSON documents over $\widetilde{\Sigma}_{\text{JSON}}$ satisfying this schema. Moreover, given an order $<$ of Σ_{key}, $\mathcal{L}_<(\mathcal{G})$ is the subset of $\mathcal{L}(\mathcal{G})$ composed of the JSON documents whose key order inside objects respects the order $<$.

We highlight that a key order over objects occurs naturally in many implementations of JSON libraries. For instance, the order $<$ can be defined as the iteration order over the data structure (such as a hash map).

Example 7.3.1. Consider again the JSON schema of Example 6.4.2. Let us order the alphabet Σ_{key} such that

$$\texttt{title} < \texttt{keywords} < \texttt{conference} < \texttt{name} < \texttt{year}.$$

The JSON document J of Example 6.4.1 belongs to $\mathcal{L}_<(\mathcal{G})$ but none of the documents equal to J up to any permutation of its key-value pairs belong to $\mathcal{L}_<(\mathcal{G})$ (permutations of the three pairs with keys `title`, `keywords`, `conference` and of the two pairs with keys `name`, `year`).

The next theorem states that, given a JSON schema, there exists a VPA $\mathcal{A}$ (resp. $\mathcal{B}$) accepting all JSON documents satisfying this schema (resp. only those respecting the key order). These two VPAs can be supposed to be minimal 1-SEVPAs by Theorem 7.2.13. The minimal 1-SEVPA $\mathcal{B}$ could be *exponentially smaller* than the minimal 1-SEVPA $\mathcal{A}$ as the order of the key-value pairs is fixed inside objects (see an illustrating example in Section B.1). We use this minimal 1-SEVPA $\mathcal{B}$ in the next section for the validation of JSON documents. The proof of the theorem is deferred to Section B.2.

Theorem 7.2.13. For any VPL L, there exists a unique minimal (with regards to the number of states) 1-SEVPA accepting L, up to a renaming of the states.

Theorem 7.3.2. *Let $\mathcal{G}$ be a closed extended CFG defining a JSON schema. Then, there exists a VPA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G})$.*
Moreover for all orders $<$ of Σ_{key}, there exists a VPA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}_<(\mathcal{G})$.

While the proof of the theorem provides a constructive approach, the construction depends on the formal semantics of JSON schemas which are still changing and being debated. Thus, to be more robust to changes in the semantics, we prefer to learn a VPA. Given an order on Σ_{key} and a JSON schema S defined by a closed extended CFG $\mathcal{G}$, the minimal 1-SEVPA accepting $\mathcal{L}_<(\mathcal{G})$ can be learned in the sense of Theorem 7.2.15. Notice that we have to adapt the learning algorithm if the teacher only knows the schema S:

- ▶ A membership query over a JSON document asks whether this document satisfies S.
- ▶ An equivalence query is answered by generating a certain number of random (valid and invalid) JSON documents and by verifying that the learned VPA $\mathcal{H}$ and the schema S agree on the documents' validity.[8]
- ▶ In membership and equivalence queries, the considered JSON documents have the order of their key-value pairs respecting the given order of Σ_{key}.

Theorem 7.2.15. Let L be a VPL over $\widetilde{\Sigma}$, n be the index of $\simeq_L$, and ζ be the length of the longest counterexample returned by the teacher for an **EQ**. Then, one can learn the minimal 1-SEVPA accepting L with

- ▶ at most $n - 1$ equivalence queries and
- ▶ a number of membership queries polynomial in n, $|\Sigma|$, and $\log \zeta$.

The randomness used in the equivalence queries implies that the learned 1-SEVPA may not exactly accept $\mathcal{L}_<(\mathcal{G})$. Setting the number of generated documents to be large helps reducing the probability that an incorrect 1-SEVPA is learned. We discuss in Section 7.5 how to generate adequate JSON documents for equivalence queries.

8: It is common to proceed this way in automata learning when exact equivalence is intractable, as explained in [Ang87, Section 4].

7.4. Streaming validation of JSON documents

As explained in the previous chapter, one important computational problem related to JSON schemas consists in determining whether a JSON document J satisfies a schema S. In this section, we provide a *streaming* algorithm that validates a JSON document against a JSON schema. By "streaming", we mean an algorithm that performs the validation test by processing the document in a single pass, symbol by symbol, and by using a limited amount of memory with respect to the size of the given document.

Our approach is new and works as follows. Given a JSON schema S, we learn the minimal 1-SEVPA $\mathcal{A}$ accepting the language $\mathcal{L}(\mathcal{A})$ equal to the set of all JSON documents J satisfying S and respecting a given order $<$ on Σ_{key}. We know that this is possible as explained in the previous section. Unfortunately, checking whether a JSON document J satisfies the JSON schema S does not amount to checking whether $J \in \mathcal{L}(\mathcal{A})$ as the key-value pairs inside the objects of J can be arbitrarily ordered. Instead, we design a streaming algorithm that uses $\mathcal{A}$ in a clever way to allow arbitrary orders of key-value pairs. To do this, we use a *key graph* defined and shown to be computable in the sequel. Then, we describe our validation algorithm and study its complexity.

Henceforth we fix a schema S given by a closed extended CFG $\mathcal{G}$, an order $<$ on Σ_{key}, and a 1-SEVPA $\mathcal{A}$ accepting $\mathcal{L}_<(\mathcal{G})$.

7.4.1. Key graph

In this section, w.l.o.g. we suppose that $\mathcal{A}$ has *no bin states*. We explain how to associate to $\mathcal{A}$ a particular graph $\mathrm{G}_{\mathcal{A}}$, called *key graph*, abstracting the stacked runs of $\mathcal{A}$ labeled by the contents of the objects appearing in words of $\mathcal{L}_<(\mathcal{G})$. More precisely, we seek every possible word $k \cdot v$ with k a key and v a valid value, *i.e.*, a primitive value, an object, or an array.[9] We then create a vertex (p, k, p') if $(p, \varepsilon) \xrightarrow{k \cdot v} (p', \varepsilon)$ is a stacked run of $\mathcal{A}$.[10] Finally, we define an edge between (p_1, k_1, p_1') and (p_2, k_2, p_2') when $p_1' \xrightarrow{\#} p_2$. That is, there exists a stacked run

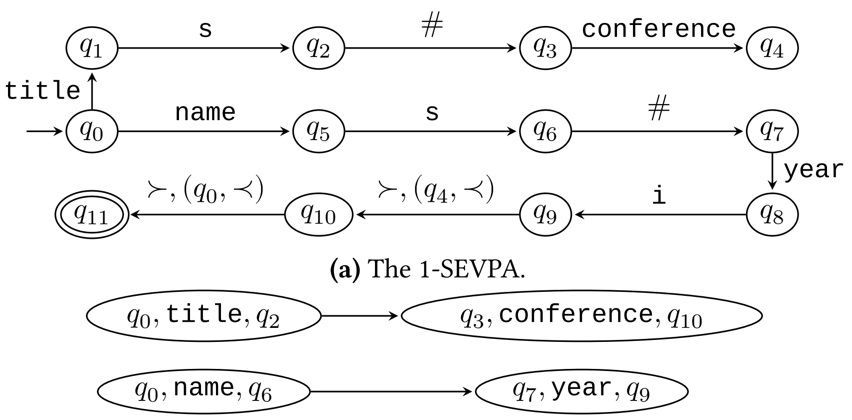
$$(p_1, \varepsilon) \xrightarrow{k_1 \cdot v_1} (p_1', \varepsilon) \xrightarrow{\#} (p_2, \varepsilon) \xrightarrow{k_2 \cdot v_2} (p_2', \varepsilon) \in \mathit{sruns}(\mathcal{A}).$$

9: Recall that any valid object or array must be a well-matched word.

10: Notice that a vertex only stores the key k and not the word $k \cdot v$.

Definition 7.4.1 (Key graph). The *key graph* $\mathrm{G}_{\mathcal{A}}$ of $\mathcal{A}$ has:

- the vertices (p, k, p') with $p, p' \in Q^{\mathcal{A}}$ and $k \in \Sigma_{\text{key}}$ if there exists a stacked run
$$(p, \varepsilon) \xrightarrow{k \cdot v} (p', \varepsilon) \in \mathit{sruns}(\mathcal{A})$$
with
$$v \in \Sigma_{\text{pVal}} \cup \{a \cdot u \cdot \bar{a} \mid a \in \Sigma_c, u \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}})\},$$
and
- the edges $((p_1, k_1, p_1'), (p_2, k_2, p_2'))$ if there exists an internal transition $p_1' \xrightarrow{\#} p_2$.



(a) The 1-SEVPA.

(b) The key graph.

Figure 7.2: A 1-SEVPA for the schema from Figure 6.3, without considering the key `keywords`, and its key graph.

By the next lemma, paths in $\mathrm{G}_{\mathcal{A}}$ focus on contents of objects being part of JSON documents satisfying S. Moreover, they abstract stacked runs of $\mathcal{A}$ in the sense that only keys k_i are stored and the subpaths labeled by the values v_i are implicit. A proof is given in Section B.3.

Lemma 7.4.2. *In a key graph $\mathrm{G}_{\mathcal{A}}$, there exists a path*

$$((p_1, k_1, p_1')(p_2, k_2, p_2') \dots (p_n, k_n, p_n'))$$

with $p_1 = q_0^{\mathcal{A}}$ if and only if there exist

- *a word $u = k_1 v_1 \# k_2 v_2 \# \dots \# k_n v_n$ such that each $k_i v_i$ is a key-value pair and u is a factor of a word in $\mathcal{L}_<(\mathcal{G})$, and*
- *a path $(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{u} (p_n', \varepsilon) \in \mathit{sruns}(\mathcal{A})$ that decomposes as follows:*

$$\forall i \in \{1, \dots, n\} : (p_i, \varepsilon) \xrightarrow{k_i v_i} (p_i', \varepsilon)$$

and

$$\forall i \in \{1, \dots, n-1\} : (p_i', \varepsilon) \xrightarrow{\#} (p_{i+1}, \varepsilon).$$

Example 7.4.3. Consider the schema from Figure 6.3 (repeated in the margin), without the key `keywords` and its associated schema. It is defined by the following closed CFG $\mathcal{G}$ (where the related productions with key permutations are not indicated, see Example 6.4.2):

$$\begin{aligned}
\mathrm{S}_0 &::= \prec\texttt{title}\ \mathrm{S}_1 \# \texttt{conference}\ \mathrm{S}_2\succ\\
\mathrm{S}_1 &::= \texttt{s}\\
\mathrm{S}_2 &::= \prec\texttt{name}\ \mathrm{S}_1 \# \texttt{year}\ \mathrm{S}_3\succ\\
\mathrm{S}_3 &::= \texttt{i}
\end{aligned}$$

A 1-SEVPA $\mathcal{A}$ accepting $\mathcal{L}_<(\mathcal{G})$ is given in Figure 7.2a. For clarity, call transitions[11] and the bin state are not represented. In Figure 7.2b, we depict

```
{
  "type": "object",
  "required": ["title",
  ↪  "conference"],
  "properties": {
    "title": { "type":
    ↪  "string" },
    "keywords": {
      "type": "array",
      "items": { "type":
      ↪  "string" }
    },
    "conference": {
      "type": "object",
      "required": ["name",
      ↪  "year"],
      "properties": {
        "name": { "type":
        ↪  "string" },
        "year": { "type":
        ↪  "integer" }
      }
    }
  }
}
```

11: Recall the particular form of call transitions and stack alphabet for 1-SEVPAs, see Definition 7.2.12.

its corresponding key graph $\mathrm{G}_{\mathcal{A}}$. Since we have the stacked run

$$(q_0, \varepsilon) \xrightarrow{\texttt{title s}} (q_2, \varepsilon) \in \mathit{sruns}(\mathcal{A}),$$

the triplet $(q_0, \texttt{title}, q_2)$ is a vertex of $\mathrm{G}_{\mathcal{A}}$. Likewise, $(q_0, \texttt{name}, q_6)$ and $(q_7, \texttt{year}, q_9)$ are vertices. Finally, as we have

$$\begin{aligned}(q_3, \varepsilon) &\xrightarrow{\texttt{conference}} (q_4, \varepsilon) \xrightarrow{\prec} (q_0, (q_4, \prec)) \\ &\xrightarrow{\texttt{name s \# year i}} (q_9, (q_4, \prec)) \\ &\xrightarrow{\succ} (q_{10}, \varepsilon) \in \mathit{sruns}(\mathcal{A}),\end{aligned}$$

the triplet $(q_3, \texttt{conference}, q_{10})$ is also a vertex of $\mathrm{G}_{\mathcal{A}}$. Finally, we have an edge from $(q_0, \texttt{title}, q_2)$ to $(q_3, \texttt{conference}, q_{10})$, since $(q_2, \varepsilon) \xrightarrow{\#} (q_3, \varepsilon)$.

From Lemma 7.4.2, we get that the key graph contains a finite number of paths, *i. e.*, it is always acyclic.

Corollary 7.4.4. *In the key graph $\mathrm{G}_{\mathcal{A}}$, there is no path $((p_1, k_1, p_1')(p_2, k_2, p_2') \dots (p_n, k_n, p_n'))$ with $p_1 = q_0^{\mathcal{A}}$ such that $k_i = k_j$ for some $i \neq j$.*

Proof. Towards a contradiction, assume the opposite. By Lemma 7.4.2, there exists

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{u} (p_n', \varepsilon) \in \mathit{sruns}(\mathcal{A})$$

such that $u = k_1 v_1 \# k_2 v_2 \# \dots \# k_n v_n$ is factor of a word in $\mathcal{L}_{<}(\mathcal{G})$. This is impossible because keys must be pairwise distinct inside objects appearing in JSON documents. □

Finally, let us discuss the size of $\mathrm{G}_{\mathcal{A}}$. While the number of vertices follow directly from the definition, the number of different paths in the graph requires a bit more work.

Lemma 7.4.5. *The key graph $\mathrm{G}_{\mathcal{A}}$ has $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^2 \cdot |\Sigma_{\text{key}}|\right)$ vertices. Moreover, visiting all vertices along all the paths of $\mathrm{G}_{\mathcal{A}}$ that start from a vertex (p, k, p') such that $p = q_0^{\mathcal{A}}$ is in*

$$\mathcal{O}\left(\left|Q^{\mathcal{A}} \times \Sigma_{\text{key}}\right|^{|\Sigma_{\text{key}}|}\right).$$

Proof. The first statement is trivial. Let us prove the second one. First, notice that a given vertex (p, k, p') has at most $\alpha = \left|\Sigma_{\text{key}} \times Q^{\mathcal{A}}\right|$ successors (q, k', q') as $\mathcal{A}$ is deterministic. Second, the length of a longest path in $\mathrm{G}_{\mathcal{A}}$ is bounded by $|\Sigma_{\text{key}}|$ by Corollary 7.4.4. Third, the number of vertices in the longest paths starting with the vertex $(q_0^{\mathcal{A}}, k, p')$ is bounded by

$$\frac{\alpha^{|\Sigma_{\text{key}}|} - 1}{\alpha - 1}.$$

As there are α potential starting vertices $(q_0^{\mathcal{A}}, k, p')$, the announced upper bound follows. □

Computing the key graph

Let us now provide a polynomial time algorithm to compute $\mathrm{G}_{\mathcal{A}}$ from $\mathcal{A}$. First, we explain how to compute the reachability relation $\mathrm{Reach}_{\mathcal{A}} \subseteq Q^{\mathcal{A}} \times Q^{\mathcal{A}}$:

- Initially, $\mathrm{Reach}_{\mathcal{A}}$ is the transitive closure of

$$\mathbb{I}_{Q^{\mathcal{A}}} \cup \{(q, q') \mid \exists (q, a, q') \in \delta_{int}\}.$$

- Repeat until $\mathrm{Reach}_{\mathcal{A}}$ does not change:
  - Add to $\mathrm{Reach}_{\mathcal{A}}$ all elements (q, q') such that there exist a call transition $q \xrightarrow{a/\gamma} p$ and a return transition $p' \xrightarrow{\bar{a}[\gamma]} q'$, with $(p, p') \in \mathrm{Reach}_{\mathcal{A}}$.
  - Close $\mathrm{Reach}_{\mathcal{A}}$ with its transitive closure.

This process terminates as the set $Q^{\mathcal{A}} \times Q^{\mathcal{A}}$ is finite. It is easy to see that $\mathrm{Reach}_{\mathcal{A}}$ can be computed in time polynomial in $|Q^{\mathcal{A}}|$ and $|\delta|$.

Second, recall that $\mathcal{A}$ can have bin states. We compute them, if there are any, and we remove them and the related transitions from $\mathcal{A}$. We provide a polynomial time algorithm to do so in Section B.4.

Finally, let us give an algorithm computing $\mathrm{G}_{\mathcal{A}}$. Its vertices are computed as follows: (p, k, p') is a vertex in $\mathrm{G}_{\mathcal{A}}$ if there exist an internal transition $p \xrightarrow{k} q$ with $k \in \Sigma_{\text{key}}$ and

- an internal transition $q \xrightarrow{a} p'$ with $a \in \Sigma_{\text{pVal}}$, or
- a call transition $q \xrightarrow{a/\gamma} r$ and a return transition $r' \xrightarrow{\bar{a}[\gamma]} p'$ with $(r, r') \in \mathrm{Reach}_{\mathcal{A}}$.

We compute the edges of $\mathrm{G}_{\mathcal{A}}$ as follows: $((p_1, k_1, p_1'), (p_2, k_2, p_2'))$ is an edge in $\mathrm{G}_{\mathcal{A}}$ if there exists an internal transition $p_1' \xrightarrow{\#} p_2$.

Therefore, the given algorithm for computing $\mathrm{G}_{\mathcal{A}}$ is polynomial, whose precise complexity is given in the next proposition and proved in Section B.4.

Proposition 7.4.6. *Computing the key graph $\mathrm{G}_{\mathcal{A}}$ is in time*

$$\mathcal{O}\left(|\delta|^2 + |\delta| \cdot |Q^{\mathcal{A}}|^4 + |Q^{\mathcal{A}}|^5 + |Q^{\mathcal{A}}|^4 \cdot |\Sigma_{\text{key}}|^2\right).$$

We highlight that the 1-SEVPA $\mathcal{A}$ and its key graph have to be computed exactly once (assuming the underlying schema does not change). That is, learning $\mathcal{A}$ and constructing $\mathrm{G}_{\mathcal{A}}$ is a *preprocessing* step of our validation algorithm, and both models can be immediately reused to validate multiple documents against the same schema.

7.4.2. Validation algorithm

In this section, we provide a streaming algorithm that validates JSON documents against a given JSON schema.

Given a word $w \in \Sigma_{\text{JSON}}^* \setminus \{\varepsilon\}$, we want to check whether $w \in \mathcal{L}(\mathcal{G})$. The main difficulty is that the key-value pairs inside an object are arbitrarily ordered in w while a fixed key order is encoded in the 1-SEVPA $\mathcal{A}$ (such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(\mathcal{G})$). Due to this, we will have to "jump around" in $\mathcal{A}$, while trying to decide whether there exists a stacked run that reads the object in the learned order. That is, we add some non-determinism within the 1-SEVPA to guess the stacked run. Our validation algorithm is inspired by the algorithm computing a deterministic VPA equivalent to some given VPA [AM04] (see Theorem 7.2.10 and its proof) and uses the key graph $\mathrm{G}_{\mathcal{A}}$ to treat arbitrary orders of the key-value pairs inside objects.

[AM04]: Alur et al. (2004), "Visibly pushdown languages"

During the reading of $w \in \Sigma_{\text{JSON}}^* \setminus \{\varepsilon\}$, in addition to checking whether $w \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}})$, the algorithm updates a subset $R \subseteq \mathrm{Reach}_{\mathcal{A}}$ and modifies the content of a stack Stk (by pushing, popping, and modifying the element on top of Stk).

First, let us explain the information stored in R. Assume that we have read the prefix zau of w such that $a \in \Sigma_c$ is the last unmatched call symbol (thus $za \in (\mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}}) \cdot \Sigma_c)^*$ and $u \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}})$).

- If a is the symbol $\sqsubset$ (*i.e.*, the start of an *ordered* array), then we have

$$R = \{(p, q) \mid (p, \varepsilon) \xrightarrow{u} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})\}.$$

- If a is the symbol $\prec$ (*i.e.*, the start of an *unordered* object), then we have

$$u = k_1 v_1 \# k_2 v_2 \# \dots k_{n-1} v_{n-1} \# u'$$

  such that $u' \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}})$ and u' is prefix of $k_n \cdot v_n$, where each $k_i \cdot v_i$ is a key-value pair. Then,

$$R = \{(p, q) \mid (p, \varepsilon) \xrightarrow{u'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})\}.$$

In the first case, by using R as defined previously, we adopt the same approach as for the determinization of VPAs. In the second case, with u, we are currently reading the key-value pairs of an object in some order that is not necessarily the one encoded in $\mathcal{A}$. In this case, the set R is focused on the key-value pair $k_n \cdot v_n$ currently being read, *i.e.*, on the word u' (instead of the whole word u). After reading of the whole object $\prec k_1 v_1 \# k_2 v_2 \# \dots \succ$, we will use the key graph $\mathrm{G}_{\mathcal{A}}$ to update the current set R.

Second, let us explain the form of the elements stored in the stack Stk. Such an element is

- either a pair $(R, \sqsubset)$,
- or a 5-tuple $(R, \prec, K, k, \mathit{Bad})$,

where

- R is a set as described previously,
- $K \subseteq \Sigma_{\text{key}}$ is a subset of keys,
- $k \in \Sigma_{\text{key}}$ is a key, and
- Bad is a set containing some vertices of $\mathrm{G}_{\mathcal{A}}$.[13]

13: In the particular case of the "empty" object $\prec\succ$, the 5-tuple $(R, \prec, K, k, \mathit{Bad})$ is replaced by $(R, \prec)$. This situation will be clarified during the presentation of our algorithm.

14: For clarity, we focus on generic VPAs first, and then specialize to 1-SEVPAs.

We are now ready to detail our streaming validation algorithm.[14] Before beginning to read w, we initialize R to the set $\mathbb{I}_{\{q_0^{\mathcal{A}}\}}$ and Stk to the empty stack. We are now going to explain how to update the current set R and the current contents of the stack Stk while reading the input word w. Suppose that we are reading the symbol a in w. In some cases, we will also peek the symbol b following a (which is a *lookahead* of one symbol).

Case (1) Suppose that a is the symbol $\sqsubset$. This means that we begin to read an array. Hence, $(R, \sqsubset)$ is pushed on Stk and R is updated to

$$R_{Upd} = \mathbb{I}_Q.$$

We thus proceed exactly as in the proof of Theorem 7.2.10.

Case (2) Suppose that $a \in \Sigma_{int}$ and $\sqsubset$ appears on top of Stk. We are thus reading the elements of an array. Hence R is updated to

$$R_{Upd} = \{(p, q) \mid \exists (p, q') \in R, q' \xrightarrow{a} q\}.$$

Again, we proceed as in the proof of Theorem 7.2.10.

Case (3) Suppose that a is the symbol $\sqsupset$. This means that we finished reading an array. If the stack is empty or its top element contains $\prec$, then $w \notin \mathcal{L}(\mathcal{G})$ and we stop the algorithm (as the word is not a valid JSON value). Otherwise $(R', \sqsubset)$ is popped from Stk and R is updated to

$$R_{Upd} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\sqsubset/\gamma} r', (r', r) \in R, r \xrightarrow{\sqsupset[\gamma]} q\},$$

as in the proof of Theorem 7.2.10.

Case (4) Suppose that a is the symbol $\prec$.

- Let us first consider the particular case where the symbol b following $\prec$ is equal to $\succ$, meaning that we will read the object $\prec\succ$. In this case, $(R, \prec)$ is pushed on Stk and R is updated to

  $$R_{Upd} = \mathbb{I}_Q$$

  as in Case (1).
- Otherwise, if b belongs to Σ_{key}, we begin to read a (non-empty) object whose treatment is different from that of an array as its key-value pairs can be read in any order. Then, R is updated to

  $$R_{Upd} = \mathbb{I}_{P_b}$$

  where

  $$P_b = \{p \in Q \mid \exists (p, b, p') \in \mathrm{G}_{\mathcal{A}}\},$$

  and $(R, \prec, K, b, Bad)$ is pushed on Stk such that
  - K is the singleton $\{b\}$ and
  - *Bad* is the empty set.

  The 5-tuple pushed on Stk indicates that the key-value pair that will be read next begins with key b; moreover $K = \{b\}$ because this is the first pair of the object. The meaning of *Bad* will be clarified later. The updated set R_{Upd} is equal to the identity relation on P_b since after reading $\prec$, we will start reading a key-value pair whose abstracted state in $\mathrm{G}_{\mathcal{A}}$ can be any state from P_b. Later while reading the object whose reading is here started, we will update the 5-tuple on top of Stk as explained below.

- Finally, it remains to consider the case where $b \notin \Sigma_{\text{key}} \cup \{\succ\}$. In this final case, we have that $w \notin \mathcal{L}(\mathcal{G})$ and we stop the algorithm.

Case (5) Suppose that $a \in \Sigma_{int} \setminus \{\#\}$ and $\prec$ appears on top of Stk. Therefore, we are currently reading a key-value pair of an object. Then, R is updated to

$$R_{Upd} = \{(p, q) \mid \exists (p, q') \in R, q' \xrightarrow{a} q'\}.$$

Case (6) Suppose that a is the symbol $\#$ and $\prec$ appears on top of Stk. This means that we just finished reading a key-value pair whose key k is stored in the 5-tuple $(R', \prec, K, k, \mathit{Bad})$ on top of Stk, and that another key-value pair will be read after symbol $\#$. The set K in $(R', \prec, K, k, \mathit{Bad})$ stores all the keys of the key-values pairs already read including k.

- If the symbol b following $\#$ does not belong to Σ_{key}, then $w \notin \mathcal{L}(\mathcal{G})$ and we stop the algorithm.
- If b belongs to K, this means that the object contains twice the same key, *i.e.*, $w \notin \mathcal{L}(\mathcal{G})$, and we also stop the algorithm.
- Otherwise, the set R is updated to

$$R_{Upd} = \mathbb{I}_{P_b},$$

with $\mathbb{I}_{P_b}$ as defined above (as we begin the reading of a new key-value pair whose key is b), and the 5-tuple $(R', \prec, K, k, \mathit{Bad})$ on top of Stk is updated such that
  - K is replaced by $K \cup \{b\}$,
  - k is replaced by b, and
  - all vertices (p, k, p') of $\mathrm{G}_{\mathcal{A}}$ such that $(p, p') \notin R$ are added to the set *Bad*.

Recall that the vertex (p, k, p') of $\mathrm{G}_{\mathcal{A}}$ is a witness of a stacked run

$$(p, \varepsilon) \xrightarrow{k \cdot v} (p', \varepsilon) \in \mathit{sruns}(\mathcal{A})$$

for some key-value pair $k \cdot v$. Hence, by adding this vertex (p, k, p') to *Bad*, we mean that the pair that has just been read does not use such a path.

Case (7) Suppose that a is the symbol $\succ$. Therefore, we end the reading of an object. If the stack is empty or its top element contains $\sqsubset$, then $w \notin \mathcal{L}(\mathcal{G})$ and we stop the algorithm. Otherwise the top of Stk contains either $(R', \prec)$ or $(R', \prec, K, k, \mathit{Bad})$ that we pop from Stk.

- If $(R', \prec)$ is popped, then we are ending the reading of the object $\prec\succ$. Hence, we proceed as in Case (3): R is updated to

$$R_{Upd} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\prec/\gamma} r' \xrightarrow{\succ[\gamma]} q\}.$$

Notice that R does not appear in R_{Upd} as $R = \mathbb{I}_Q$.
- If $(R', \prec, K, k, \mathit{Bad})$ is popped, we are reading the last key-value pair with key k and we add to *Bad* all vertices (p, k, p') of $\mathrm{G}_{\mathcal{A}}$ such that $(p, p') \notin R$ as done in Case (6). Next, let $\mathrm{Valid}(K, \mathit{Bad})$ be

the set of pairs of states $(r, r') \in Q^2$ such that there exists a path $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ in $\mathrm{G}_{\mathcal{A}}$ with

- $p_1 = r$, $p'_n = r'$,
- $(p_i, k_i, p'_i) \notin \mathit{Bad}$ for all $i \in \{1, \dots, n\}$,
- $K = \{k_1, k_2, \dots, k_n\}$.

Then, R is updated to:

$$R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\prec/\gamma} r', \\ (r', r) \in \mathrm{Valid}(K, \mathit{Bad}), r \xrightarrow{\succ[\gamma]} q\}.$$

We thus proceed as in Case (3) except that condition $(r', r) \in R$ is replaced by $(r', r) \in \mathrm{Valid}(K, \mathit{Bad})$. With the latter condition, we check that the key-value pairs that have been read as composing an object of w are such that the same pairs ordered as imposed by $\mathcal{A}$ label some stacked run of $\mathcal{A}$, *i. e.*, the corresponding abstracted path appears in $\mathrm{G}_{\mathcal{A}}$.

Case (8) Suppose that $a \in \Sigma_{\mathit{int}}$ and Stk is empty, then $w \notin \mathcal{L}(\mathcal{G})$ and we stop the algorithm. Indeed an internal symbol appears either in an array or in an object (see Cases (2), (5), and (6)).

Finally, when the input word w is completely read, we check whether the stack Stk is empty and the computed set R contains a pair $(q_0^{\mathcal{A}}, q)$ with $q \in F$.

Notice that the previous algorithm is supposed to have a 1-SEVPA as input, for which all the call transitions (q, a, q', γ) are such that $q' = q_0^{\mathcal{A}}$.[15] Therefore, some simplifications can be done in the algorithm, namely:

15: See Definition 7.2.12.

- ▶ in Case (1), R is updated to $R_{\mathit{Upd}} = \mathbb{I}_{\{q_0^{\mathcal{A}}\}}$,
- ▶ in Case (3), R is updated to

$$R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\sqsubset/\gamma} q_0^{\mathcal{A}}, (q_0^{\mathcal{A}}, r) \in R, r \xrightarrow{\sqsupset[\gamma]} q\},$$

- ▶ in Case (7), if $(R', \prec)$ is popped, then

$$R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\prec/\gamma} q_0^{\mathcal{A}} \xrightarrow{\succ[\gamma]} q\},$$

otherwise R is updated to

$$R_{\mathit{Upd}} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\prec/\gamma} q_0^{\mathcal{A}}, \\ (q_0^{\mathcal{A}}, r) \in \mathrm{Valid}(K, \mathit{Bad}), r \xrightarrow{\succ[\gamma]} q\}.$$

In particular, the computation of $\mathrm{Valid}(K, \mathit{Bad})$ can be restricted to pairs (r, r') such that $r = q_0^{\mathcal{A}}$.

The resulting algorithm is given in Algorithm 7.1. This algorithm does not include the *preprocessing algorithm* of learning a 1-SEVPA $\mathcal{A}$ from the JSON schema S and of computing the key graph $\mathrm{G}_{\mathcal{A}}$.

Finally, the next theorem, whose proof is deferred to Section B.5, gives the time and memory complexities of the algorithm.

Algorithm 7.1: Streaming algorithm for validating JSON documents against a given JSON schema.

```
Require: A 1-SEVPA $\mathcal{A}$ over $\widetilde{\Sigma}_{\text{JSON}}$ accepting $\mathcal{L}_{<}(\mathcal{G})$ for a closed extended CFG $\mathcal{G}$ defining a JSON schema, its key graph $\mathrm{G}_{\mathcal{A}}$, and a word $w \in \Sigma_{\text{JSON}}^{*} \setminus \{\varepsilon\}$.
Ensure: yes is returned if and only if $w \in \mathcal{L}(\mathcal{G})$.
$(R, Stk) \leftarrow (\mathbb{I}_{\{q_0^{\mathcal{A}}\}}, \varepsilon); a \leftarrow w_1$    ▷ $w_i$ is the $i$th symbol of $w$
for all $i \in \{2, \dots, |w|\}$ do
    if $i \leq |w|$ then $b \leftarrow w_i$ else $b \leftarrow \varepsilon$
    if $a \in \Sigma_c$ then
        if $a = \sqsubset$ then Push$(Stk, (R, \sqsubset)); R \leftarrow \mathbb{I}_{\{q_0^{\mathcal{A}}\}}$
        else    ▷ As $a \in \Sigma_c$, we have $a = \prec$
            if $b = \succ$ then Push$(Stk, (R, \prec)); R \leftarrow \mathbb{I}_{\{q_0^{\mathcal{A}}\}}$
            else if $b \in \Sigma_{\text{key}}$ then Push$(Stk, (R, \prec, \{b\}, b, \emptyset)); R \leftarrow \mathbb{I}_{P_b}$
            else return no
    else if $a \in \Sigma_r$ then
        if $a = \sqsupset$ (resp. $\succ$) and there is no $\sqsubset$ (resp. $\prec$) on top of $Stk$ then return no
        if $a = \sqsupset$ then
            $(R', \sqsubset) \leftarrow$ Pop$(Stk)$
            $R \leftarrow \{(p, q) \mid (p, p') \in R', p' \xrightarrow{\sqsubset/\gamma} q_0, (q_0, r) \in R, r \xrightarrow{\sqsupset[\gamma]} q\}$
        else if ($a = \succ$ and $(R', \prec)$ is on top of $Stk$ for some $R'$) then
            $(R', \prec) \leftarrow$ Pop$(Stk)$
            $R \leftarrow \{(p, q) \mid (p, p') \in R', p' \xrightarrow{\prec/\gamma} q_0^{\mathcal{A}} \xrightarrow{\succ[\gamma]} q\}$
        else
            $(R', \prec, K, k, Bad) \leftarrow$ Pop$(Stk)$
            $Bad \leftarrow Bad \cup \{(p, k, p') \in \mathrm{G}_{\mathcal{A}} \mid (p, p') \notin R\}$
            $V \leftarrow$ Valid$(K, Bad)$
            $R \leftarrow \{(p, q) \mid (p, p') \in R', p' \xrightarrow{\prec/\gamma} q_0^{\mathcal{A}}, (q_0, r) \in V, r \xrightarrow{\succ[\gamma]} q\}$
    else if $a \in \Sigma_{int}$ then
        if ($\sqsubset$ appears on top of $Stk$) or ($a \neq \#$ and $\prec$ appears on top of $Stk$) then
            $R \leftarrow \{(p, q) \mid (p, p') \in R, p' \xrightarrow{a} q\}$
        else if $a = \#$ and $\prec$ appears on top of $Stk$ then
            $(R', \prec, K, k, Bad) \leftarrow$ Pop$(Stk)$
            if ($b \notin \Sigma_{\text{key}}$ or $b \in K$) then return no
            $K \leftarrow K \cup \{b\}$
            $Bad \leftarrow Bad \cup \{(p, k, p') \in \mathrm{G}_{\mathcal{A}} \mid (p, p') \notin R\}$
            Push$(Stk, (R', \prec, K, b, Bad)); R \leftarrow \mathbb{I}_{P_b}$
        else return no
    $a \leftarrow b$
if ($Stk = \varepsilon$ and there exists $(q_0, q) \in R$ with $q \in F$) then return yes else return no
```

Theorem 7.4.7. *Let S be a JSON schema defined by a closed extended CFG $\mathcal{G}$ and $\mathcal{A}$ be a 1-SEVPA $\mathcal{A}$ accepting $\mathcal{L}_<(\mathcal{G})$. Then, checking whether a JSON document J with depth $depth(J)$ satisfies the schema S*

- *is in time*
$$\mathcal{O}\left(|J| \cdot \left(|Q|^4 + |Q|^{|\Sigma_{\text{key}}|} \cdot |\Sigma_{\text{key}}|^{|\Sigma_{\text{key}}|+1}\right)\right),$$
- *and uses an amount of memory in*
$$\mathcal{O}\left(|\delta| + |\mathcal{A}|^2 \cdot |\Sigma_{\text{key}}| + depth(J) \cdot \left(|\mathcal{A}|^2 + |\Sigma_{\text{key}}|\right)\right).$$

7.5. Implementation and experiments

In this section, we discuss the Java implementation of our framework. That is, we implemented a way to learn a 1-SEVPA $\mathcal{A}$ from a JSON schema, the construction of the key graph $\mathrm{G}_{\mathcal{A}}$, and our validation algorithm.

Recall that we introduced in Section 6.5 the "classical" validation algorithm, used in many implementations. In order to match the abstractions we defined (see Section 6.4) and to have options to tune the learning process, we implemented our own classical validator. This implementation includes some optimizations:

- If an object does not contain all the required keys, we immediately return false, without needing to validate each key-value pair.
- If an object or an array has too few or too many elements (with regards to the constraints defined in the schema), we also immediately return false.

Our goal in this section is to compare this classical approach and our streaming algorithm. Before performing this comparison, we explain how one can implement a teacher for the learning framework, *i.e.*, a teacher that has a JSON schema and has to decide whether a JSON document is valid for the schema, and whether a 1-SEVPA accepts the target language. In particular, Section 7.5.1 explains how to generate random JSON documents, while Section 7.5.2 focuses on the queries of the teacher. Then, Section 7.5.3 discusses the experimental framework and the results. Finally, Section 7.5.4 studies a worst-case scenario for the classical algorithm.

The reader is referred to the code documentation for more details about our implementation.[16]

16: See `https://github.com/DocSkellington/automatalib`, `https://github.com/DocSkellington/learnlib`, `https://github.com/DocSkellington/JSONSchemaTools`, and `https://github.com/DocSkellington/ValidatingJSONDocumentsWithLearnedVPA`. A Zenodo artifact is available at `https://zenodo.org/records/7309690`.

In the remaining of this section, let us assume we have a JSON schema S_0.

7.5.1. Generating JSON documents

The learning process of a 1-SEVPA from a JSON schema S_0 requires to generate JSON documents that satisfy or do not satisfy S_0 (see Section 7.3). We briefly explain in this section the generators that we implemented, first for generating valid documents and then for generating invalid documents.

Valid documents

If the grammar $\mathcal{G}$ defining the S_0 does not contain any Boolean operations, generating a document satisfying the schema is easy by following the semantics of $\mathcal{G}$ as explained in Section 6.4.1. Let us roughly explain how the generation works when $\mathcal{G}$ contains Boolean operations and S_0 is *not recursive*.[17] We follow the productions of $\mathcal{G}$ in a top-down manner as follows. If the current production is a disjunction

17: The recursive schemas will be discussed later.

$$\mathrm{S} ::= \mathrm{S}_1 \vee \mathrm{S}_2 \vee \cdots \vee \mathrm{S}_n,$$

then we select one S_i and replace this production by $\mathrm{S} ::= \mathrm{S}_i$. It may happen that the chosen S_i leads to no valid JSON document. Thus, we may need to try multiple S_i before successfully generating a valid document. In case of a production

$$\mathrm{S} ::= \mathrm{S}_1 \wedge \mathrm{S}_2 \wedge \cdots \wedge \mathrm{S}_n$$

or $\neg S_1$, then we propagate these Boolean operations lower in the grammar by replacing each S_i by its production and by rewriting the resulting right-hand side as a disjunction. For instance, if

$$\mathrm{S} ::= \mathrm{S}_1 \wedge \mathrm{S}_2$$

with $\mathrm{S}_1 ::= \mathtt{i}$ and $\mathrm{S}_2 ::= \prec k \mathrm{S}_3 \succ \vee\, \mathtt{s}$, then we get

$$\mathrm{S} ::= (\mathtt{i} \wedge \prec k \mathrm{S}_3 \succ) \vee (\mathtt{i} \wedge \mathtt{s}).$$

In this simple example, no choice in the disjunction leads to a valid document. In general, the propagation of Boolean operations may require several iterations before yielding a JSON document. We implemented two types of generator, supporting all Boolean operations:

- a *random* generator where each choice in disjunctions is made at random, and
- an *exhaustive* generator that explores every choice, thus producing every valid document one by one.

Invalid documents

We also implemented modifications of these generators to allow the creation of invalid documents. The idea is to follow the same algorithm as before but to sometimes *deviate* from the grammar $\mathcal{G}$. For instance, if the current production describes an integer, we can decide to instead generate an array or a string. Moreover, among a predefined finite set of possible deviations, we can either choose one randomly or exhaustively explore each of them, in a way similar to mutation testing [DLS78; JH11]. As for valid documents, we respectively call random and exhaustive those two generators.

[DLS78]: DeMillo et al. (1978), "Hints on Test Data Selection: Help for the Practicing Programmer"
[JH11]: Jia et al. (2011), "An Analysis and Survey of the Development of Mutation Testing"

Maximal depth for recursive schemas

We also allow the user to set a *maximal depth* (*i.e.*, the maximal number of nested objects or arrays). When a generator reaches the maximal depth, it can no longer produce objects or arrays. This is useful for recursive schemas or when generating invalid documents, as it permits us to be sure we eventually produce a document of finite depth.

Support for enumerations and regular expressions for keys

It is noteworthy that the implementation supports an abstracted version of enumerations of JSON values inside documents. Recall that, for instance, all strings are abstracted by `s`. In a similar way, an enumeration is abstracted by `e`. While this implies that the exact values are lost, it allows us to keep the enumerations inside the considered schemas. Moreover, if the schemas contains regular expressions to define the keys inside an object,[18] the regular expression is directly used as the key. That is, we do not generate a string that matches the expression but uses the expression itself as the key.

18: Using the JSON schema field called `patternProperties`.

7.5.2. Learning algorithm and queries

Let us now focus on the learning algorithm itself, and in particular on the membership and equivalence queries. We recall that the equivalence queries are performed by generating a certain number of (valid and invalid) JSON documents and by verifying that the learned VPA $\mathcal{H}$ and the given schema S_0 agree on the documents' validity. As said in Section 7.3, we use the $\mathtt{TTT}_{\mathrm{VPL}}$ algorithm [Isb15] to learn a 1-SEVPA from S_0. More precisely, we rely on the implementation made by Isberner in the well-known Java libraries AutomataLib [19] and LearnLib [20].

[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"

19: `https://learnlib.de/pages/automatalib`

20: `https://learnlib.de/`

Using the fact that the membership and equivalence queries can be defined independently from the actual learning algorithm, we implement the queries as follows. We use the random and exhaustive generators of valid and invalid documents as explained in Section 7.5.1 and we fix two constants C and D depending on the schema to be learned.[21] For a *membership* query over a word $w \in \Sigma^*_{\mathrm{JSON}}$, the teacher runs the classical validator described in Algorithm 6.1 on w and S_0. For an *equivalence* query over a learned 1-SEVPA $\mathcal{H}$, the teacher performs multiple checks, in this order:

21: The values of C and D are given in Section 7.5.3.

1. Is there a loop $(q_0^{\mathcal{H}}, a, q_0^{\mathcal{H}})$ over the initial state of $\mathcal{H}$ reading an internal symbol $a \in \Sigma_{int}$?[22] In that case, we generate a word w accepted by $\mathcal{H}$.[23] The word aw is then a counterexample since it is also accepted by $\mathcal{H}$ but it is not a valid document (it is not an object).
2. Using a (random or exhaustive) generator for valid documents, is there a valid document w that is not accepted by $\mathcal{H}$? In that case, w is a counterexample. If the used generator is the exhaustive one, we generate every possible valid document one by one over the course of all equivalence queries. We stop using it once all documents have been generated. If the generator is the random one, for each equivalence query,

22: We observed such a situation several times from our experimentation.

23: This is supported by LearnLib.

```
{
    "type": "object",
    "properties": {
        "name": { "type": "string" },
        "children": {
            "type": "array",
            "maxItems": 1,
            "items": { "$ref": "#" }
        }
    },
    "required": ["name"],
    "additionalProperties": false
}
```

Figure 7.3: The recursive JSON schema.

we generate C valid documents for each document depth between 0 and D.

3. Using a (random or exhaustive) generator for invalid documents, is there an invalid document w that is accepted by $\mathcal{H}$? In that case, w is a counterexample. Both generators are used as explained in the previous step.
4. In the key graph $\mathrm{G}_{\mathcal{H}}$, is there a path
$$((p_1, k_1, p_1')(p_2, k_2, p_2') \dots (p_n, k_n, p_n'))$$
with $p_1 = q_0^{\mathcal{H}}$ such that $k_i = k_j$ for some $i \neq j$ (see Corollary 7.4.4)? In that case, from this path, we construct a counterexample, *i. e.*, a word accepted by $\mathcal{H}$ that is not a valid document. See Section B.6 for the details.

Corollary 7.4.4. In the key graph $\mathrm{G}_{\mathcal{A}}$, there is no path $((p_1, k_1, p_1')(p_2, k_2, p_2') \dots (p_n, k_n, p_n'))$ with $p_1 = q_0^{\mathcal{A}}$ such that $k_i = k_j$ for some $i \neq j$.

If $\mathcal{H}$ fails every test (*i. e.*, we could not find a counterexample), we conclude that $\mathcal{H}$ is correct, the equivalence query succeeds, and we finish the learning process.

7.5.3. Experimental evaluation

We now discuss our experimental results, starting with a description of the six schemas we considered. We then give, for each schema, the time and memory needed to learn a 1-SEVPA and to construct its key graph. Finally, we compare the classical validation algorithm presented in Section 6.5 and ours, on several randomly generated JSON documents, some of which being valid, while the others are invalid.

Evaluated schemas

For the experimental evaluation of our algorithms, we consider the following schemas, sorted in increasing size:

1. A schema that accepts documents defined recursively. Each object contains a string and can contain an array whose single element satisfies the whole schema, *i. e.*, this is a recursive list. See Figure 7.3.

```
{
    "Title": "Fast",
    "type": "object",
    "required": ["string", "double", "integer", "boolean", "object", "array"],
    "additionalProperties": false,
    "properties": {
        "string": { "type": "string" },
        "double": { "type": "number" },
        "integer": { "type": "integer" },
        "boolean": { "type": "boolean" },
        "object": {
            "type": "object",
            "required": ["anything"],
            "additionalProperties": false,
            "properties": {
                "anything": {
                    "type": ["number", "integer", "boolean", "string"]
                }
            }
        },
        "array": {
            "type": "array",
            "items": { "type": "string" },
            "minItems": 2
        }
    }
}
```

Figure 7.4: The JSON schema accepting documents with all types.

2. A schema that accepts documents containing each type of values, *i.e.*, an object, an array, a string, a number, an integer, and a Boolean. See Figure 7.4.
3. A schema that defines how snippets must be described in *Visual Studio Code*.[24]
4. A recursive schema that defines how the metadata files for *VIM plugins* must be written.[25]
5. A schema that defines how *Azure Functions Proxies* files must look like.[26]
6. A schema that defines the configuration file for a code coverage tool called *codecov*.[27]

That is, we consider two schemas written by ourselves to test our framework, and four schemas that are used in real world cases. The last four schemas were modified to make all object keys mandatory[28] and to remove unsupported keywords. All schemas and scripts used for the benchmarks can be consulted on our repository.[29] In the rest of this section, the schemas are referred to by their order in the enumeration.

We present three types of experimental results. First, we discuss the time and the number of membership and equivalence queries needed to learn a 1-SEVPA from a JSON schema. Then, given such a 1-SEVPA $\mathcal{A}$, we give the time and memory required to compute its reachability relation $\mathrm{Reach}_{\mathcal{A}}$ and its key graph $\mathrm{G}_{\mathcal{A}}$. Finally, the time and memory used to validate a JSON document using the classical and our new algorithms are provided. The server used for the benchmarks ran Debian 10 over Linux 5.4.73-1-pve with a 4-core Intel® Xeon® Silver 4214R Processor with 16.5M cache, and 64GB of RAM. Moreover,

24: This schema can be downloaded from `https://raw.githubusercontent.com/Yash-Singh1/vscode-snippets-json-schema/main/schema.json`

25: `https://json.schemastore.org/vim-addon-info.json`

26: `https://json.schemastore.org/proxies.json`

27: `https://json.schemastore.org/codecov.json`

28: That is, we added all the keys in a `required` field in each object.

29: `https://github.com/DocSkellington/ValidatingJSONDocumentsWithLearnedVPA`

Time (s)	Membership	Equivalence	$\|Q\|$	$\|\Sigma\|$	$\|\delta_c\|$	$\|\delta_r\|$	$\|\delta_i\|$	Diameter
2.2	2055.0	5.0	7	15	14	3.0	5.0	3.0
4.5	69514.0	3.0	24	20	48	3.0	26.0	12.0
9.0	21943.0	5.0	16	17	32	7.0	18.0	13.0
9590.3	4246085.0	36.4	150	27	300	2946.5	760.3	9.0
35008.2	4063971.7	30.5	121	35	242	2123.0	752.5	13.3
Timeout	633049534.0	192.0	884	77	1768	89695.0	8557.0	28.0

Table 7.1: Results of the learning benchmarks. For the first five schemas, values are averaged out of ten experiments. For the last schema, only one experiment was conducted.

we used OpenJDK version 11.0.12.

Learning VPAs

As the first part of the preprocessing step of our validation algorithm, we must learn a 1-SEVPA. For the first three evaluated schemas, we use an exhaustive generator thanks to the small number of valid documents that can be produced (up to a certain depth D defined below). For the remaining three schemas, we rely instead on a random generator where we fix the number of generated documents per round at $C = 10000$. For both exhaustive and random generators, the maximal depth of the generated documents is set at $D = \mathit{depth}(S) + 1$, where $\mathit{depth}(S)$ is the maximal number of nested defined objects and arrays in the schema S, except for the recursive list schema where $D = 10$, and for the recursive *VIM plugins* schema where $D = 7$.

In all cases, we learn the 1-SEVPA while measuring the total time (including the time needed for the membership and equivalence queries), and the total number of queries. After removing the bin state of the resulting 1-SEVPA (see Section B.4), we also measure its number of states and transitions, as well as its diameter. For the first five schemas, we do not set a time limit and repeat the learning process ten times. For the last schema, we set a time limit of one week and, for time constraints, only perform the learning process once. After that, we stop the computation and retrieve the learned 1-SEVPA at that point.

Recall that for one of the checks performed in an equivalence query,[30] we must construct the key graph of the hypothesis $\mathcal{H}$. In particular, we must compute its reachability relation $\mathrm{Reach}_{\mathcal{H}}$. As we observe that a counterexample implies only a small change in $\mathcal{H}$, we gain efficiency by avoiding to compute the new reachability relation from scratch at each equivalence query.

30: Check Item 4. in Section 7.5.2.

The results for the learning benchmarks are given in Table 7.1. The learning process for the last schema was not completed after one week. The retrieved learned 1-SEVPA is therefore an approximation of this schema. Its key graph has repeated keys along some of its paths, a situation that cannot occur if the 1-SEVPA was correctly learned, see Corollary 7.4.4. Notice that, when using a random generator, different runs of the algorithm may yield different 1-SEVPAs.

Corollary 7.4.4. In the key graph $G_{\mathcal{A}}$, there is no path $((p_1, k_1, p_1')(p_2, k_2, p_2') \dots (p_n, k_n, p_n'))$ with $p_1 = q_0^{\mathcal{A}}$ such that $k_i = k_j$ for some $i \neq j$.

$\mathrm{Reach}_{\mathcal{A}}$			$\mathrm{G}_{\mathcal{A}}$			
Time (s)	Memory (kB)	Size	Time (s)	Computation (kB)	Storage (kB)	Size
34	492	31	100	2231	65	3
67	1152	213	234	2623	69	9
67	737	125	118	2223	69	10
1756	10316	5832	1715	11827	419	418
2208	13978	4420	2839	17968	667	541
377141	212970	270886	187659	120398	16335	6397

Table 7.2: Time and memory needed to compute $\mathrm{Reach}_{\mathcal{A}}$ and $\mathrm{G}_{\mathcal{A}}$, and their size. Values are taken from a single experiment. For $\mathrm{G}_{\mathcal{A}}$, the Computation (resp. Storage) column gives the memory required to compute $\mathrm{G}_{\mathcal{A}}$ (resp. to store $\mathrm{G}_{\mathcal{A}}$).

Construction of the key graph

The second part of the preprocessing step is to construct the key graph of the learned 1-SEVPA. For each evaluated schema, we select the learned 1-SEVPA with the largest set of states, in order to report a worst-case measure. From that 1-SEVPA $\mathcal{A}$, we compute its reachability relation $\mathrm{Reach}_{\mathcal{A}}$ and its key graph $\mathrm{G}_{\mathcal{A}}$, and measure the time and memory used, as well as their sizes. Values obtained after a single experiment are given in Table 7.2. We can see that the size of the key graph is far from the theoretical upper bound in $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^2 \cdot \left|\Sigma_{\mathrm{key}}\right|\right)$ of Lemma 7.4.5. Moreover, its storage does not consume more than one megabyte, except for *codecov* schema. That is, even for non-trivial schemas, the key graph is relatively lightweight.

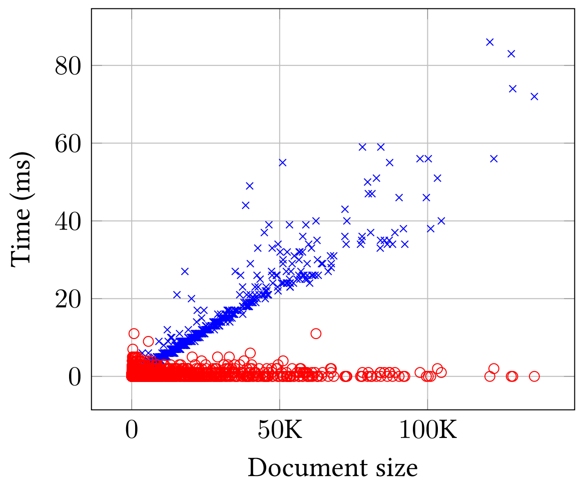
Lemma 7.4.5. The key graph $\mathrm{G}_{\mathcal{A}}$ has $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^2 \cdot \left|\Sigma_{\mathrm{key}}\right|\right)$ vertices. Moreover, visiting all vertices along all the paths of $\mathrm{G}_{\mathcal{A}}$ that start from a vertex (p, k, p') such that $p = q_0^{\mathcal{A}}$ is in $\mathcal{O}\left(\left|Q^{\mathcal{A}} \times \Sigma_{\mathrm{key}}\right|^{\left|\Sigma_{\mathrm{key}}\right|}\right)$.

Comparing validation algorithms

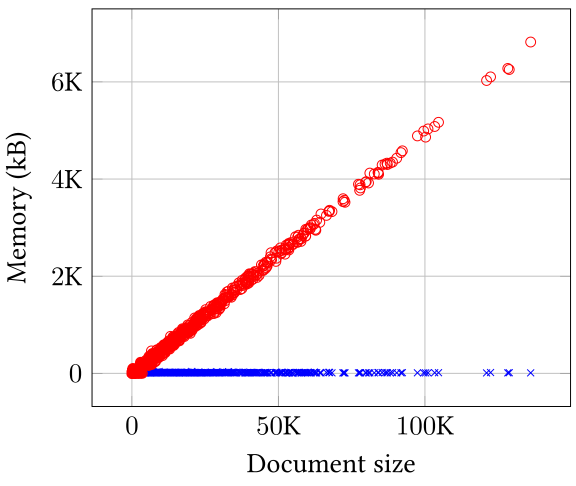
Finally, we compare both validation algorithms: the classical one and our new streaming algorithm. For the latter, we use the 1-SEVPA (and its key graph) selected in the previous section. We first generate 5000 valid and 5000 invalid JSON documents using a random generator, with a maximal depth equal to $D = 20$. We then execute both validation algorithms on these documents, while measuring the time and memory required. On all considered documents, both algorithms return the same classification output, even for the partially learned 1-SEVPA.

Since obtaining a close approximation of the consumed memory requires Java to "stop the world" to destroy all unused objects, we execute each algorithm twice: one time where we only measure time, and a second time where we request the memory to be cleaned each iteration of the algorithm (or each recursive call).

For our algorithm, we only measure the memory required to execute the algorithm, as we do not need to store the whole document to be able to process it. We also do not count the memory to store the 1-SEVPA and its key graph. As the classical algorithm must have the complete document stored in memory, in this case, we sum the RAM consumption for the document and for the algorithm itself. This is coherent to what happens in actual web-service



(a) Time.



(b) Memory.

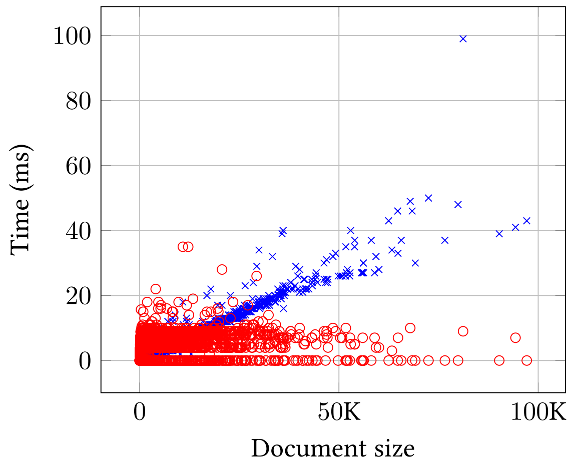
Figure 7.5: Results of validation benchmarks for the metadata files for *VIM plugins*. Blue crosses give the values for our algorithm, and red circles the values for the classical validator. Values are averaged out of ten experiments.

handling: Whenever a new validation request is received, we would spawn a new subprocess that handles a specific document. Since the 1-SEVPA and its key graph are the same for all subprocesses, they would be loaded in a memory space shared by all processes.

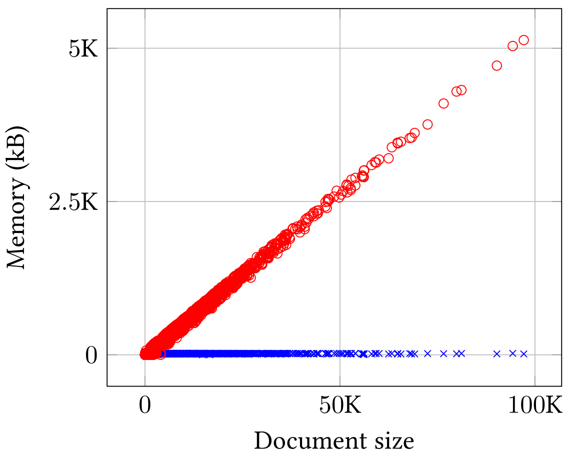
In order to focus on the three largest schemas, we defer the presentation of the first three schemas to Section B.7. Results for *VIM plugins*, *Azure Functions Proxies*, and *codecov* are given in Figures 7.5 to 7.7. The blue crosses give the values for our algorithm, while the red circles stand for the classical algorithm. The x-axis gives the number of symbols of our alphabet in each document. We can see that our algorithm usually requires more time than the classical algorithm to validate a document. We observed that a majority of the total time is spent computing the set $\text{Valid}(K, \mathit{Bad})$. For both *VIM plugins* and *Azure Functions Proxies*, our algorithm consumes less memory than the classical algorithm. We observed that, even if we add the number of bytes needed to store the automaton and the key graph (see Tables 7.1 and 7.2), we remain under the classical algorithm, especially for large documents.

For the last *codecov* schema, we recall that the learning process was not completed, leading to an approximated 1-SEVPA whose key graph has repeated keys. We had to modify our validation algorithm, in particular the computation of the set $\text{Valid}(K, \mathit{Bad})$:[31] when visiting each path of the key graph, we pay attention to stop it on the second visit of a key. This means that we have to explore some invalid paths, increasing the memory and time consumed by our algorithm. Thus, it appears that, while a 1-SEVPA that is not completely learned can still be used in our algorithm to correctly decide whether a document is valid, stopping the learning process early may drastically increase the time and space required.

31: See Section B.5.

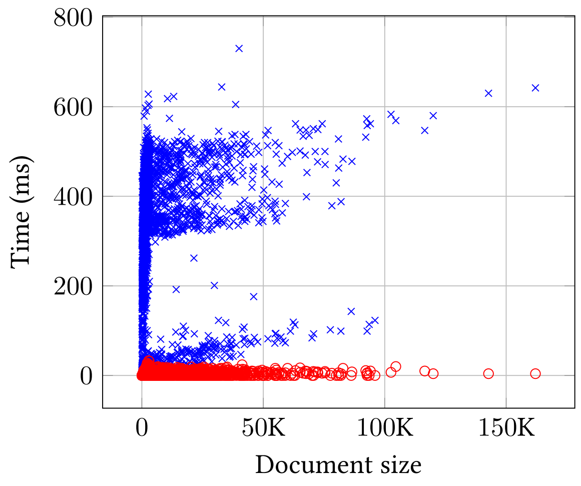


(a) Time.

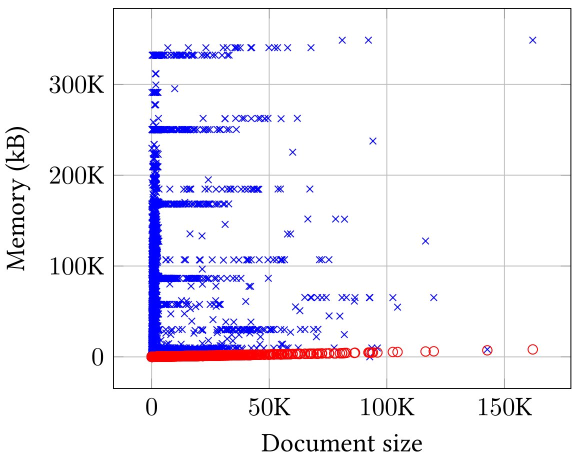


(b) Memory.

Figure 7.6: Results of validation benchmarks for the *Azure Functions Proxies* file.



(a) Time.



(b) Memory.

Figure 7.7: Results of validation benchmarks for the configuration files of *codecov*.

7.5.4. Worst case for the classical validator

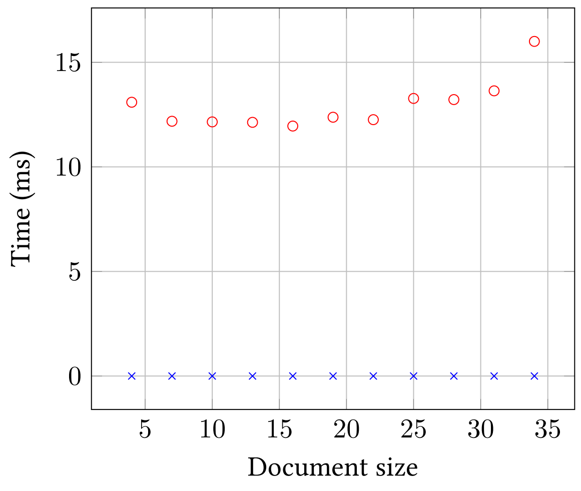
Finally, let us consider one last schema, purposefully made to highlight the differences between both algorithms. Its extended CFG (up to key permutations) is as follows, with $\ell \geq 1$:

$$\begin{aligned} \mathsf{S} &::= \mathsf{S}_1 \wedge \mathsf{S}_2 \wedge \cdots \wedge \mathsf{S}_\ell \\ \forall i \in \{1, \dots, \ell\} : \mathsf{S}_i &::= R_i \vee R_{i+1} \vee \cdots \vee R_\ell \\ \forall i \in \{1, \dots, \ell\} : R_i &::= \prec k_i \mathsf{s} \# k_{i+1} \mathsf{s} \# \dots \# k_\ell \mathsf{s} \succ \end{aligned}$$

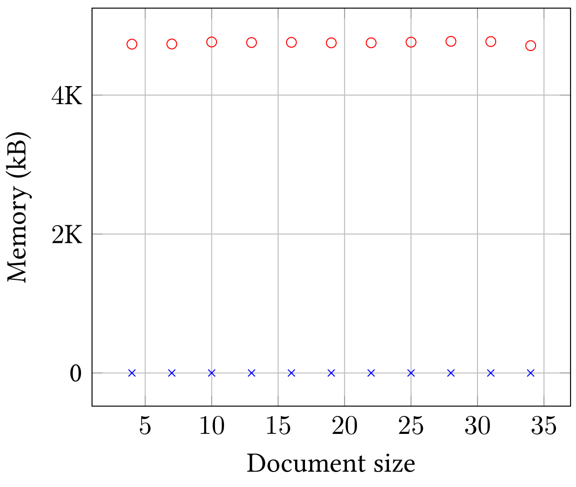
Notice the difference between S_i and S_{i+1}: R_i is removed from S_i to get S_{i+1}. Hence an equivalent grammar is $\mathsf{S} ::= R_\ell$. On the one hand, the classical validator has to explore each S_i in the conjunction, and each R_j in the related disjunction, before finally considering S_ℓ and thus R_ℓ. On the other hand, as we learn minimal 1-SEVPAs, we can here easily construct a 1-SEVPA directly for the simplified grammar $\mathsf{S} ::= R_\ell$. This example is a worst-case for the

Reach$_{\mathcal{A}}$			G$_{\mathcal{A}}$			
Time (s)	Memory (kB)	Size	Time (s)	Computation (kB)	Storage (kB)	Size
33.0	492.0	31.0	109.0	1981.0	60.0	2.0

Table 7.3: Time and memory needed to compute Reach$_{\mathcal{A}}$ and G$_{\mathcal{A}}$, and their size. Values are taken from a single experiment. For G$_{\mathcal{A}}$, the Computation (resp. Storage) column gives the memory required to compute G$_{\mathcal{A}}$ (resp. to store G$_{\mathcal{A}}$).



(a) Time.



(b) Memory.

Figure 7.8: Results of validation benchmarks for the *worst-case* schema, with $\ell = 10$.

classical validator, while being easy to handle for our algorithm, as shown in Table 7.3 and Figure 7.8.

7.6. Conclusion

In this chapter, we have proved that, given any JSON schema, one can construct a VPA that accepts the same set of JSON documents as the schema. Leveraging this fact and TTT, we designed a learning algorithm that yields a VPA for the schema, under a fixed order of the keys inside objects. We then abstracted as a key graph the part of this VPA dealing with objects, and proposed a streaming algorithm that uses both the graph and the VPA to decide whether a document is valid for the schema, under any order on the keys.

As future work, one could focus on constructing the VPA directly from the schema, without going through a learning algorithm. While this task is easy if the schema does not contain Boolean operations, it is not yet clear how to proceed in the general case. Second, it could be worthwhile to compare our algorithm against an implementation of a classical algorithm used in the industry. This would require either to modify the industrial implementations to support abstractions, or to modify our algorithm to work on unabstracted JSON schemas. Third, in our validation approach, we decided to use a VPA accepting the JSON documents satisfying a fixed key order — thus requiring to use the key graph and its costly computation of the set Valid(K, Bad). It could be interesting to make additional experiments to compare this approach with one where we instead use a VPA accepting the JSON documents and all their key permutations — in this case, reasoning on the key graph would no longer be needed.

Furthermore, motivated by obtaining efficient querying algorithms on XML trees, the authors of [SSM08] have introduced the concept of mixed automata in a way to accept subsets of unranked trees where some nodes have ordered sons and some other have unordered sons. It would be interesting to adapt our validation algorithm to different formalisms of documents, such as the one of mixed automata.

[SSM08]: Seidl et al. (2008), "Counting in trees"

Finally, exploring the possibility of using *systems of procedural automata* (SPAs, for short) [FS21], which form an extension of DFAs that can mutually call each other. The restrictions imposed by Frohme and Steffen for the learning algorithm (namely, that calls and returns between the different DFAs are observable) make the model akin to VPAs. As SPAs have specific structures (*i. e.*, distinct DFAs with special transitions indicating how to jump between them) and as the learning algorithm proposed in [FS21] has a lower time complexity than $\texttt{TTT}_{\mathrm{VPL}}$, it may be interesting to adapt the ideas from our validation algorithm to SPAs, potentially yielding an algorithm with a better time complexity. Notably, Frohme and Steffen applied their algorithm to infer automata encoding Document Type Definitions (DTDs) which are used to encode constraints over XML documents [FS18].

[FS21]: Frohme et al. (2021), "Compositional learning of mutually recursive procedural systems"

[FS18]: Frohme et al. (2018), "Active Mining of Document Type Definitions"

Technical Proofs and Details of Chapter 7 B.

This chapter, based on [BPS23], contains the technical details and proofs that were not given in Chapter 7. That is, it serves as the appendix of the previous chapter.

[BPS23]: Bruyère et al. (2023), "Validating Streaming JSON Documents with Learned VPAs"

Chapter contents

B.1. Interest of fixing a key order

In this section, we provide a family of JSON schemas S_n, $n \geq 1$, and a key-order, such that the minimal 1-SEVPA $\mathcal{A}_n$ accepting all the JSON documents satisfying S_n is exponentially larger than the minimal 1-SEVPA $\mathcal{B}_n$ accepting those documents respecting the key order.

Let $\Sigma_{\mathsf{key}} = \{k_1, \dots, k_n\}$ with the key order $k_1 < \dots < k_n$. The proposed JSON schema S_n is the one defining all JSON documents

$$\prec k_{i_1} \mathsf{s} \# \dots \# k_{i_n} \mathsf{s} \succ$$

where $(k_{i_1}, \dots, k_{i_n})$ is a permutation of $(k_1, \dots, k_n)$. It is easy to see that the 1-SEVPA $\mathcal{B}_n$ has $\mathcal{O}(n)$ states since it only accepts the document

$$\prec k_1 \mathsf{s} \# \dots \# k_n \mathsf{s} \succ.$$

Let us show that the 1-SEVPA $\mathcal{A}_n$ has at least 2^n states.

Let us consider the state q of $\mathcal{A}_n$ such that

$$(q_0, \varepsilon) \xrightarrow{\prec} (q_0, \gamma) \xrightarrow{x} (q, \gamma) \xrightarrow{y} (p, \gamma) \xrightarrow{\succ} (r, \varepsilon) \in \mathit{sruns}(\mathcal{A})$$

with $r \in F$, $x = k_{i_1} \mathsf{s} \# \dots \# k_{i_\ell} \mathsf{s}$, and $y = k_{i_{\ell+1}} \mathsf{s} \# \dots \# k_{i_n} \mathsf{s}$ where $(k_{i_1}, \dots, k_{i_\ell}, \dots, k_{i_n})$ is a permutation of $(k_1, \dots, k_n)$. It is not possible to have another stacked run $(q_0, \gamma) \xrightarrow{x'} (q, \gamma)$ of $\mathcal{A}$ with $x' = k_{j_1} \mathsf{s} \# \dots \# k_{j_m} \mathsf{s}$ such that $\{k_{i_1}, \dots, k_{i_\ell}\} \neq \{k_{j_1}, \dots, k_{j_m}\}$. Otherwise, $\mathcal{A}_n$ would accept an invalid document. It follows that there are as many such states q as there are subsets $\{k_{i_1}, \dots, k_{i_\ell}\}$ of Σ_{key}. Therefore $\mathcal{A}_n$ has at least 2^n states.

B.2. Proof of Theorem 7.3.2

Theorem 7.3.2. *Let $\mathcal{G}$ be a closed extended CFG defining a JSON schema. Then, there exists a VPA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G})$.*
Moreover for all orders $<$ of Σ_{key}, there exists a VPA $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}_{<}(\mathcal{G})$.

In order to prove this theorem, we first apply multiple modifications to the grammar. Then, we will construct a VPA from the resulting grammar and show that this VPA is equivalent to the grammar.

Let $\mathcal{G}$ be a closed extended CFG defining a JSON schema. To prove the existence of a VPA $\mathcal{A}$ accepting $\mathcal{L}(\mathcal{G})$, we adapt a construction provided in the proof of [KMV07, Theorem 1].[1] Recall that

[KMV07]: Kumar et al. (2007), "Visibly pushdown automata for streaming XML"

1: Some difficulties arise from the presence of Boolean operators in the grammar productions.

$$\Sigma_c = \{\prec, \sqsubset\},$$
$$\Sigma_r = \{\succ, \sqsupset\},$$

and

$$\Sigma_{int} = \Sigma_{\text{pVal}} \cup \Sigma_{\text{key}} \cup \{\#\}.$$

The terminal alphabet of $\mathcal{G}$ is equal to the (pushdown) alphabet

$$\widetilde{\Sigma}_{\text{JSON}} = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}.$$

Suppose that its non-terminal alphabet is equal to

$$\mathcal{S} = \{\mathsf{S}_0, \mathsf{S}_1, \dots, \mathsf{S}_n\}$$

where S_0 is the axiom. We assume that any production that has S_0 as left-hand side is of the form $S_0 ::= \prec e \succ$, where e is a generalized regular expression over the alphabet $\mathcal{S} \cup \Sigma_{int}$, as JSON documents are supposed to be objects. We need to apply several steps in a way to construct the required VPA $\mathcal{A}$.

(i) We transform the grammar $\mathcal{G}$ such that:

- the left-hand sides of productions are pairwise distinct,
- each production is of the form $\mathsf{S}_j ::= a_j e_j \bar{a}_j$ such that $a_j \in \{\prec, \sqsubset\}$ and e_j is a generalized regular expression over the alphabet $\mathcal{S} \cup \Sigma_{int}$.

We proceed as follows to obtain the transformed grammar.

1. If S_j appears as the left-hand side of $k \geq 2$ productions, then it is replaced by k new non-terminal symbols $\mathsf{S}_{j_1}, \mathsf{S}_{j_2}, \dots, \mathsf{S}_{j_k}$ (one for each of the k productions), and each of occurrence of S_j in the right-hand side of productions is replaced by $\mathsf{S}_{j_1} \vee \mathsf{S}_{j_2} \vee \dots \vee \mathsf{S}_{j_k}$.
2. Any production $\mathsf{S}_j ::= v$ with $v \in \Sigma_{\text{pVal}}$ is deleted. Each occurrence of S_j in the right-hand side of productions is replaced by v.
3. We proceed as in the previous item with all productions $\mathsf{S}_j ::= \mathsf{S}_{j_1} \vee \mathsf{S}_{j_2} \vee \dots \vee \mathsf{S}_{j_n}$, $\mathsf{S}_j ::= \mathsf{S}_{j_1} \wedge \mathsf{S}_{j_2} \wedge \dots \wedge \mathsf{S}_{j_n}$, and $\mathsf{S}_j ::= \neg \mathsf{S}_{j_1}$.

After this transformation, a non-terminal S_j uniquely identifies a production and the latter is of the form $\mathsf{S}_j ::= a_j e_j \bar{a}_j$ with $a_j \in \{\prec, \sqsubset\}$.[2] Notice that the axiom S_0 of the original grammar may have been replaced by several axioms.

2: This process terminates because the grammar is well-formed (see Remark 6.4.5).

Given a production $\mathrm{S}_j ::= \prec e_j \succ$, the generalized regular expression e_j is of the form $k_1\phi_1 \,\#\, k_2\phi_2 \,\#\, \dots \,\#\, k_n\phi_n$ where each ϕ_i is a Boolean expression of elements from $\Sigma_{\mathrm{pVal}} \cup \mathcal{S}$. Similarly, the expression e_j in each production $\mathrm{S}_j ::= \sqsubset e_j \sqsupset$ is of the form either $\varepsilon \vee \phi_1(\#\phi_1)^*$, or $\phi_1 \,\#\, \dots \,\#\, \phi_1$.

(ii) In a way to obtain a normal form for the productions

$$\mathrm{S}_j ::= a_j e_j \bar{a}_j,$$

we simplify the Boolean expressions ϕ_i that appear in the generalized regular expressions e_j, and the productions containing them, as follows.

1. Each Boolean expression ϕ_i is put into disjunctive normal form (DNF). We will see that 0 (meaning *false*) may appear inside ϕ_i that is thus simplified with the usual Boolean rules.
2. In each DNF ϕ_i,
   - each conjunction that contains some $v, v' \in \Sigma_{\mathrm{pVal}}$, with $v \neq v'$, is replaced by 0,
   - each conjunction that contains some $v \in \Sigma_{\mathrm{pVal}}$ and $S \in \mathcal{S}$ is replaced by 0 (indeed S defines either an object or an array),
   - each conjunction that contains some $S, S' \in \mathcal{S}$ is replaced by 0, whenever S defines an object and S' an array, or the contrary.
3. Every production $\mathrm{S}_j ::= a_j e_j \bar{a}_j$ such that e_j contains a Boolean expression ϕ_i equal to 0 is replaced by $\mathrm{S}_j ::= 0$, with one exception detailed hereafter. Each occurrence of S_j in the right-hand side of other productions is thus replaced by 0. The exception is

   $$\mathrm{S}_j ::= \sqsubset \varepsilon \vee \phi_1(\#\phi_1)^* \sqsupset$$

   which is simplified into

   $$\mathrm{S}_j ::= \sqsubset \varepsilon \sqsupset$$

   when ϕ_1 is equal to 0.

After this second step, symbol 0 does not appear in any DNF formula ϕ_i and some productions may have the form $\mathrm{S}_j ::= 0$.

In the next steps (iii) and (iv), we want to modify the grammar such that operators $\neg$ and $\wedge$ disappear from the DNF formulas ϕ_i.

(iii) The simplified DNF formulas ϕ_i have literals of the form v $\neg v$, S_j, or $\neg \mathrm{S}_j$, with $v \in \Sigma_{\mathrm{key}}$, $\mathrm{S}_j \in \mathcal{S}$. We want to define each $\neg v$ and $\neg \mathrm{S}_j$ by some additional new productions, in a way to replace them by those productions.

1. For this purpose, we first add the productions defining the set of all JSON values as given in Lemma 6.4.4 (we recall that these productions use the non-terminal U).
2. Let us explain how to define $\neg v$, with $v \in \Sigma_{\mathrm{pVal}}$, by adequate new productions. We need to define all JSON values except v. This is possible by adapting the grammar of Lemma 6.4.4 and by using the non-terminal U:
   - $R_1 ::= v'$ for all $v' \in \Sigma_{\mathrm{pVal}} \setminus \{v\}$,
   - $R_2 ::= \prec k_1 U \,\#\, k_2 U \,\#\, \dots \,\#\, k_n U \succ$ for all sequences $(k_1, \dots, k_n)$, $n \geq 0$, of pairwise distinct keys of Σ_{key},

Lemma 6.4.4. For a given set of keys Σ_{key}, the set of all valid JSON values (*i.e.*, primitive values or well-formed objects and arrays) J is equal to $\mathcal{L}(\mathcal{G}_{\mathrm{U}})$ where $\mathcal{G}_{\mathrm{U}}$ is the closed extended CFG given in Figure 6.5, called *universal*, with U being the axiom.

- ▶ $R_3 ::= \sqsubset\varepsilon \vee U(\# U)^*\sqsupset$,
- ▶ $\mathsf{S}_{\neg v} ::= R_1 \vee R_2 \vee R_3$.

We can then replace each occurrence of $\neg v$ in the DNF formulas ϕ_i by $\mathsf{S}_{\neg v}$.

3. Let us now explain how to define $\neg\mathsf{S}_j$, with $\mathsf{S}_j \in \mathcal{S}$, by adequate productions. Recall that S_j is the left-hand side of the production $\mathsf{S}_j ::= a_j e_j \bar{a}_j$. Let us consider the particular example $a_j e_j \bar{a}_j = \prec k_1\phi_1 \# k_2\phi_2 \succ$ (the reader could infer the general case from this particular example). We need to define all JSON values except the ones defined by S_j. Defining $\neg S_j$ is done thanks to the following grammar:
   - ▶ $T_1 ::= \prec k'_1 U \# k'_2 U \# \dots \# k'_n U \succ$ for all sequences $(k'_1, \dots, k'_n)$, $n \geq 0$, of pairwise distinct keys of Σ_{key}, except sequence (k_1, k_2),
   - ▶ $T_1 ::= \prec k_1 \neg\phi_1 \# k_2\phi_2 \succ$
   - ▶ $T_1 ::= \prec k_1 \phi_1 \# k_2\neg\phi_2 \succ$
   - ▶ $T_1 ::= \prec k_1 \neg\phi_1 \# k_2\neg\phi_2 \succ$
   - ▶ $T_2 ::= v$ for all $v \in \Sigma_{\text{pVal}}$,
   - ▶ $T_3 ::= \sqsubset\varepsilon \vee U(\# U)^*\sqsupset$,
   - ▶ $\mathsf{S}_{\neg \mathsf{S}_j} ::= T_1 \vee T_2 \vee T_3$.

   We can then replace each occurrence of $\neg S_j$ in the DNF formulas ϕ_i by $\mathsf{S}_{\neg \mathsf{S}_j}$.

With this step, several new productions have been added to the productions $S_j ::= a_j e_j \bar{a}_j$, with $S_j \in \mathcal{S}$. We again apply to those new productions the transformations described in (i) and (ii), and we replace each occurrence of $\neg v$ and $\neg\mathsf{S}_j$ by $\mathsf{S}_{\neg v}$ and $\mathsf{S}_{\neg S_j}$ respectively. Notice there is no occurrence of neither $\neg U$, nor $\neg R_j$, nor $\neg T_j$, $j = 1, 2, 3$, in the new productions,[3] and, therefore, no need to define them by some productions. After this step, no production contains the negation of a primitive value or a non-terminal. We use the same notation $\mathcal{S}$ for the set of all non-terminals.

3: The used Boolean operators are limited to $\vee$

(iv) We now proceed to simplify the conjunctions away. For each set of non-terminals $\{S_{i_1}, \dots, S_{i_k}\} \subseteq \mathcal{S}$, we want to define $\varphi = S_{i_1} \wedge \dots \wedge S_{i_k}$ by an additional new production with a left-hand side denoted by S_φ. We have the following cases:

1. If the set $\{S_{i_1}, \dots, S_{i_k}\}$ contains some non-terminals defining objects and some others defining arrays, then we define $\mathsf{S}_\varphi ::= 0$.
2. If the set contains only non-terminals defining objects, we can assume $S_{i_\ell} ::= \prec k_1^\ell \phi_1^\ell \# \dots \# k_{m_\ell}^\ell \phi_{m_\ell}^\ell \succ$. We have two possibilities:
   - ▶ If all S_{i_ℓ} use the same sequence $(k_1, \dots, k_m)$ of keys, then we set $\mathsf{S}_\varphi ::= \prec k_1(\wedge_{\ell=1}^k \phi_1^\ell) \# \dots \# k_m(\wedge_{\ell=1}^k \phi_m^\ell)\succ$.
   - ▶ Otherwise, we set $\mathsf{S}_\varphi ::= 0$.
3. Suppose that the set contains only non-terminals defining arrays.
   - ▶ In the case where all arrays have a fixed number of elements, we perform as in the previous item.
   - ▶ If these arrays all define an unbounded number of elements, *i.e.*, $\mathsf{S}_{i_\ell} ::= \sqsubset\varepsilon \vee \phi^\ell(\#\phi^\ell)^*\sqsupset$, then we set

$$\mathsf{S}_\varphi ::= \sqsubset\varepsilon \vee (\wedge_{\ell=1}^k \phi^\ell)(\#(\wedge_{\ell=1}^k \phi^\ell))^*\sqsupset.$$

- We now focus on the case where some non-terminals define a fixed number m of elements (which we can assume is the same for each non-terminal; otherwise, let $\mathsf{S}_\varphi ::= 0$), and some other non-terminals define an unbounded number of elements. Then, we set

$$\mathsf{S}_\varphi ::= \sqsubset(\wedge_{\ell=1}^{k}\phi^\ell) \# \dots \# (\wedge_{\ell=1}^{k}\phi^\ell)\sqsupset,$$

  such that we define m elements.

Once the new productions S_φ are defined for all conjunctions $\varphi = S_{i_1} \wedge \dots \wedge S_{i_k}$, we replace by S_φ every occurrence of φ in the DNF formulas of the productions. Moreover, as the new productions with left-hand side S_φ may contain occurrences of formulas of the form $\wedge_{\ell=1}^{k}\phi^\ell$, we put those formulas in DNF and we also replace the resulting conjunctions φ' by the non-terminal $\mathsf{S}_{\varphi'}$. In this way, no production contains anymore neither negations nor conjunctions.

(v) Finally, here is the last step to obtain the required VPA $\mathcal{A}$ accepting $\mathcal{L}(\mathcal{G})$. We have a series of productions in our grammar, whose some have a right-hand side equal to 0. We delete those productions. For the remaining ones, we rename by $\mathcal{S} = \{\mathsf{S}_1, \dots, \mathsf{S}_n\}$ the resulting alphabet of non-terminals, and by $\mathsf{S}_j ::= a_j e_j \bar{a}_j$ the production whose $\mathsf{S}_j \in \mathcal{S}$ is the unique left-hand side, for all $j \in \{1, \dots, n\}$. We partition $\mathcal{S}$ into $\mathcal{S}^{\prec} \cup \mathcal{S}^{\sqsubset}$, such that $\mathsf{S}_j \in \mathcal{S}^{a_j}$ according to the value of a_j in the production $\mathsf{S}_j ::= a_j e_j \bar{a}_j$. Recall that the axiom of the original grammar $\mathcal{G}$ may have been replaced by several axioms in $\mathcal{S}$ by step (i). Recall also that the DNF formulas appearing inside each e_j are now disjunctions of symbols from $\mathcal{S} \cup \Sigma_{int}$, *i.e.*, e_j is a *classical* regular expression over $\mathcal{S} \cup \Sigma_{int}$ (*i.e.*, a regular expression using union, concatenation and Kleene-$*$ operation).[4]

4: Steps $(ii) - (iv)$ are new compared to the proof of [KMV07, Theorem 1], due to the presence of Boolean operators $\neg$ and $\wedge$ in the grammar productions.

It may happen that $\mathcal{S}$ contains no axiom S_j (since each production $\mathsf{S}_j ::= 0$, with S_j being an axiom, has been deleted). In this case, it is easy to construct a VPA accepting the empty language. For the sequel, we thus suppose that this situation does not hold.

For each production $\mathsf{S}_j ::= a_j e_j \bar{a}_j$, we construct a complete DFA $\mathcal{B}_j$ over the alphabet $\mathcal{S} \cup \Sigma_{int}$ for the (classical) regular expression e_j. Specifically, let

$$\mathcal{B}_j = (\mathsf{S} \cup \Sigma_{int}, Q^{\mathcal{B}_j}, q_0^{\mathcal{B}_j}, q_0^{\mathcal{B}_j}, \delta^{\mathcal{B}_j})$$

be that DFA. If this automaton has bin states, we suppose that it is unique and denoted by $\bot_j$. We then construct the cartesian product $\mathcal{B}$ of all the automata $\mathcal{B}_j$, $j \in \{1, \dots, n\}$.

We are finally ready to construct the VPA $\mathcal{A}$, which is more precisely a 1-SEVPA over $\widetilde{\Sigma}_{\mathsf{JSON}}$ and using $\Gamma = Q^{\mathcal{A}} \times \{\prec, \sqsubset\}$ as its stack alphabet. We define $\mathcal{A}$ such that:

- Its set of states is the set of states of $\mathcal{B}$ with a new state q_f which is the unique final state of $\mathcal{A}$. Moreover, the initial state of $\mathcal{A}$ is composed of the initial state of each $\mathcal{B}_j$. That is,

$$\begin{aligned} Q^{\mathcal{A}} &= \{q_f\} \uplus (Q^{\mathcal{B}_1} \times Q^{\mathcal{B}_2} \times \dots \times Q^{\mathcal{B}_n}) \\ q_0^{\mathcal{A}} &= (q_0^{\mathcal{B}_1}, \dots, q_0^{\mathcal{B}_n}) = q_0^{\mathcal{B}} \\ F^{\mathcal{A}} &= \{q_f\}. \end{aligned}$$

- ▶ The set of internal transitions $\delta_{int}^{\mathcal{A}}$ are the transitions of $\mathcal{B}$ labeled by symbols in Σ_{int} (and, thus, not in $\mathcal{S}$).
- ▶ For each transition $p \xrightarrow{\mathrm{S}_j} p' \in runs(\mathcal{B})$ where $\mathrm{S}_j \in \mathrm{S}^a$, with $a \in \{\prec, \sqsubset\}$, we add the following transitions in $\mathcal{A}$:
  - the call transition $p \xrightarrow{a/(p,a)} q_0^{\mathcal{A}}$ in $\delta_c^{\mathcal{A}}$,[5] and
  - the return transitions $q \xrightarrow{\bar{a}[(p,a)]} p'$ in $\delta_r^{\mathcal{A}}$ for all $q = (q_1, \dots, q_n) \in Q^{\mathcal{A}}$ such that $q_j \in F^{\mathcal{B}_j}$.[6]
- ▶ For each $j \in \{1, \dots, n\}$ such that S_j is an axiom, we add the call transition $q_0^{\mathcal{A}} \xrightarrow{\prec/(q_0^{\mathcal{A}},\prec)} q_0^{\mathcal{A}}$ to $\delta_c^{\mathcal{A}}$ and the return transition $q_0^{\mathcal{A}} \xrightarrow{\succ[(q_0^{\mathcal{A}},\prec)]} q_f$ to $\delta_r^{\mathcal{A}}$ for all $q = (q_1, \dots, q_n) \in Q^{\mathcal{A}}$ such that $q_j \in F^{\mathcal{B}_j}$.

5: After reading a, we jump to the initial state of $\mathcal{A}$ and push (p, a) on the stack

6: At the end of a run in $\mathcal{B}$ that is accepting in $\mathcal{B}_j$, with (p, a) on top of the stack, we go to the state p' after reading $\bar{a}$.

Notice that the constructed 1-SEVPA may be non deterministic by the way the return transitions have been defined (it is deterministic when we only consider the internal and call transitions).

Let us prove the following lemma, stating that $\mathcal{A}$ accepts $\mathcal{L}(\mathcal{G})$. This will be sufficient to conclude the proof of Theorem 7.3.2. Indeed, from that lemma, we can derive that if S_j is an axiom of $\mathcal{G}$, then $aw\bar{a}$ satisfies S_j if and only if the stacked run for w ends in a state $(q_1, \dots, q_n)$ in which $q_j \in F^{\mathcal{B}_j}$. By construction of $\mathcal{A}$, this is equivalent to reach the final state q_f after reading $aw\bar{a}$ from the initial state $q_0^{\mathcal{A}}$. Hence we conclude that the language of $\mathcal{A}$ is the union of the languages described by each axiom, *i.e.*, $\mathcal{A}$ accepts $\mathcal{L}(\mathcal{G})$.

That is, once the lemma is proved, we have the first statement of Theorem 7.3.2. The theorem also states that given a grammar $\mathcal{G}$ and an order $<$ over Σ_{key}, a VPA $\mathcal{A}$ can be constructed such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(\mathcal{G})$. The construction is exactly the same as done above, by taking into account the fixed order imposed on Σ_{key} in the productions defining objects.

Lemma B.2.1. *For any non-terminal $\mathrm{S}_j \in \mathcal{S}^a$ (with $a \in \{\prec, \sqsubset\}$) and any word $aw\bar{a} \in \mathrm{WM}(\widetilde{\Sigma})$, it holds that $aw\bar{a} \in \mathcal{L}(\mathrm{S}_j)$ if and only if there exists a state $q = (q_1, \dots, q_n) \in Q^{\mathcal{A}}$ such that $q_j \in F^{\mathcal{B}_j}$ and $(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w} (q, \varepsilon) \in sruns(\mathcal{A})$.*

Proof. To ease the writing of the proof, let us introduce a notation to designate the language of a disjunction over $\mathcal{S}$:

$$\mathcal{L}(\mathrm{S}_{i_1} \vee \dots \vee \mathrm{S}_{i_n}) = \bigcup_{j=1}^{n} \mathcal{L}(\mathrm{S}_{i_j}),$$

where each S_{i_j} is a non-terminal of $\mathcal{G}$ and $\mathcal{L}(\mathrm{S}_{i_j})$ is the set of words that satisfy S_{i_j}. We also write $\mathrm{S} \in \phi$ when the non-terminal S is present in the disjunction ϕ.

Let $a \in \{\prec, \sqsubset\}$, $S_j \in \mathcal{S}^a$ be a non-terminal of $\mathcal{G}$, and $aw\bar{a} \in \mathrm{WM}(\widetilde{\Sigma})$. We prove our lemma by induction over the depth of w (recall that the depth of w is the maximal number of unmatched call symbols among the prefixes of w).

Base case. The depth of w is zero, *i.e.*, $w \in \Sigma^*_{int}$. By construction of $\mathcal{A}$, we have

$$\begin{aligned} & (q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}) \\ \Leftrightarrow \quad & q_0^{\mathcal{A}} \xrightarrow{w} q \in \mathit{runs}(\mathcal{B}). \end{aligned}$$

So, by construction of $\mathcal{B}$, $w \in \mathcal{L}(\mathrm{S}_j)$ if and only if $q_j \in F^{\mathcal{B}_j}$.[7]

7: Recall that $q_0^{\mathcal{A}} = q_0^{\mathcal{B}}$.

Induction step. Let $d \in \mathbb{N}$ and assume the lemma holds for every word of depth lower or equal to d. Let $w \in \mathrm{WM}(\widetilde{\Sigma})$ of depth $d+1$. We suppose that the production of S_j is of the shape

$$\mathrm{S}_j ::= au_1\phi_1 \dots u_m\phi_m u_{m+1}\bar{a} \tag{B.2.i}$$

with each $u_\ell \in \Sigma^*_{int}$ and ϕ_j is a disjunction over non-terminals of $\mathcal{S}$. Notice that this covers the definition of objects, and of arrays of shape $\sqsubset\phi_1 \# \dots \# \phi_1\sqsupset$. The case for the second definition of arrays (where $\mathrm{S}_j ::= \sqsubset\varepsilon \vee \phi_1(\#\phi_1)^*\sqsupset$) can be handled in a similar fashion. We prove the equivalence stated in the lemma by showing both directions.

Membership implies stacked run. Assume $aw\bar{a} \in \mathcal{L}(\mathrm{S}_j)$. We want to prove that there is a run in $\mathcal{A}$ reading w that ends in a state $q = (q_1, \dots, q_n)$ such that $q_j \in F^{\mathcal{B}_j}$.
As $aw\bar{a} \in \mathcal{L}(\mathrm{S}_j)$, see (B.2.i), we can decompose

$$w = u_1 b_1 w_1 \bar{b}_1 \dots u_m b_m w_m \bar{b}_m u_{m+1}$$

such that $b_\ell w_\ell \bar{b}_\ell \in \mathcal{L}(\phi_\ell)$ and w_ℓ has depth $\leq d$, for each $\ell \in \{1, \dots, m\}$. That is, there exists a non-terminal $\mathrm{S}_{h_\ell} \in \phi_\ell$ such that $b_\ell w_\ell \bar{b}_\ell \in \mathcal{L}(\mathrm{S}_{h_\ell})$. By the induction hypothesis, we thus have states $r^1, \dots, r^m$ such that $(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w_\ell} (r^\ell, \varepsilon) \in \mathit{sruns}(\mathcal{A})$ and $r^\ell_{h_\ell} \in F^{\mathcal{B}_{h_\ell}}$ for each $\ell \in \{1, \dots, m\}$. Then, we have the following stacked run in $\mathcal{A}$, with $p^\ell, s^\ell \in Q$ and $\alpha_\ell \in \Gamma$

$$\begin{aligned} (q_0^{\mathcal{A}}, \varepsilon) & \xrightarrow{u_1} (p^1, \varepsilon) \xrightarrow{b_1} (q_0^{\mathcal{A}}, \alpha_1) \xrightarrow{w_1} (r^1, \alpha_1) \xrightarrow{\bar{b}_1} (s^1, \varepsilon) \xrightarrow{u_2} \dots \\ & \xrightarrow{u_m} (p^m, \varepsilon) \xrightarrow{b_m} (q_0^{\mathcal{A}}, \alpha_m) \xrightarrow{w_m} (r^m, \alpha_m) \xrightarrow{\bar{b}_m} (s^m, \varepsilon) \\ & \xrightarrow{u_{m+1}} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}) \end{aligned}$$

with the following equivalent run in $\mathcal{B}$

$$q_0^{\mathcal{A}} \xrightarrow{u_1} p^1 \xrightarrow{\mathrm{S}_{h_1}} s^1 \xrightarrow{u_2} \dots \xrightarrow{u_m} p^m \xrightarrow{\mathrm{S}_{h_m}} s^m \xrightarrow{u_{m+1}} q \in \mathit{runs}(\mathcal{B}).$$

By construction of $\mathcal{B}$, as $aw\bar{a} \in \mathcal{L}(\mathrm{S}_j)$, it must be that

$$u_1 \mathrm{S}_{h_1} u_2 \dots u_m \mathrm{S}_{h_m} u_{m+1} \in \mathcal{L}(\mathcal{B}_j),$$

which is equivalent to $q_j \in F^{\mathcal{B}_j}$.

Stacked run implies membership. Assume there is a state $q = (q_1, \dots, q_n)$ such that $q_j \in F^{\mathcal{B}_j}$ and $(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w} (q, \varepsilon)$. We show that $aw\bar{a} \in \mathcal{L}(\mathrm{S}_j)$. We decompose

$$w = u'_1 b'_1 w'_1 \bar{b}'_1 \dots u'_t b'_t w'_t \bar{b}'_t u'_{t+1}$$

with $t \geq 1$ (as the depth of w is positive), $u'_\ell \in \Sigma^*_{int}$, and $b'_\ell w'_\ell \bar{b}'_\ell \in \mathrm{WM}(\widetilde{\Sigma})$ for each ℓ. Thus, we can decompose the given run into

$$\begin{aligned}(q_0^{\mathcal{A}}, \varepsilon) &\xrightarrow{u'_1} (p^1, \varepsilon) \xrightarrow{b'_1} (q_0^{\mathcal{A}}, \alpha_1) \xrightarrow{w'_1} (r^1, \alpha_1) \xrightarrow{\bar{b}'_1} (s^1, \varepsilon) \xrightarrow{u'_2} \dots \\ &\xrightarrow{u'_t} (p^t, \varepsilon) \xrightarrow{b'_t} (q_0^{\mathcal{A}}, \alpha_t) \xrightarrow{w'_t} (r^t, \alpha_t) \xrightarrow{\bar{b}'_t} (s^t, \varepsilon) \\ &\xrightarrow{u'_{t+1}} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})\end{aligned}$$

As $q_j \in F^{\mathcal{B}_j}$, it must be that $q_j \neq \bot_j$. By construction of $\mathcal{A}$ and $\mathcal{B}$, we then have $p^\ell_j \neq \bot_j$ and $s^\ell_j \neq \bot_j$ for every $\ell \in \{1, \dots, t\}$. Moreover, for each $\ell \in \{1, \dots, t\}$, there must exist a non-terminal S_{h_ℓ} such that $r^\ell_{h_\ell} \in F^{\mathcal{B}_{h_\ell}}$. We thus have a corresponding run in $\mathcal{B}$

$$q_0^{\mathcal{A}} \xrightarrow{u'_1} p^1 \xrightarrow{\mathrm{S}_{h_1}} s^1 \xrightarrow{u'_2} \dots \xrightarrow{u'_t} p^t \xrightarrow{\mathrm{S}_{h_t}} s^t \xrightarrow{u'_{t+1}} q \in \mathit{runs}(\mathcal{B})$$

By the induction hypothesis, it follows that $b'_\ell w'_\ell \bar{b}'_\ell \in \mathcal{L}(\mathrm{S}_{h_\ell})$ for all ℓ. Recall that, by our assumption, the latter run, seen in $\mathcal{B}_j$, leads to the final state $q_j \in F^{\mathcal{B}_j}$. That is, $u'_1 \mathrm{S}_{h_1} \dots u'_t \mathrm{S}_{h_t} u'_{t+1}$ belongs to $\mathcal{L}(\mathcal{B}_j)$. By (B.2.i), this means that $t = m$, each u'_ℓ is equal to u_ℓ, and each S_{h_ℓ} belongs to ϕ_ℓ. Therefore $aw\bar{a}$ belongs to $\mathcal{L}(\mathrm{S}_j)$. The lemma is thus proved. As stated before, we then deduce that $\mathcal{A}$ accepts $\mathcal{L}(\mathcal{G})$. □

B.3. Proof of Lemma 7.4.2

Lemma 7.4.2. *In a key graph $\mathrm{G}_{\mathcal{A}}$, there exists a path*

$$((p_1, k_1, p'_1)(p_2, k_2, p'_2) \dots (p_n, k_n, p'_n))$$

with $p_1 = q_0^{\mathcal{A}}$ if and only if there exist

- *a word $u = k_1 v_1 \# k_2 v_2 \# \dots \# k_n v_n$ such that each $k_i v_i$ is a key-value pair and u is a factor of a word in $\mathcal{L}_<(\mathcal{G})$, and*
- *a path $(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{u} (p'_n, \varepsilon) \in \mathit{sruns}(\mathcal{A})$ that decomposes as follows:*

$$\forall i \in \{1, \dots, n\} : (p_i, \varepsilon) \xrightarrow{k_i v_i} (p'_i, \varepsilon)$$

and

$$\forall i \in \{1, \dots, n-1\} : (p'_i, \varepsilon) \xrightarrow{\#} (p_{i+1}, \varepsilon).$$

We only prove one implication, the other being easily proved. Suppose that there exists a path

$$((p_1, k_1, p'_1)(p_2, k_2, p'_2) \dots (p_n, k_n, p'_n))$$

in $\mathrm{G}_{\mathcal{A}}$ with $p_1 = q_0$. Then, by definition of $\mathrm{G}_{\mathcal{A}}$, there exists a stacked run of $\mathcal{A}$

$$(q_0, \varepsilon) \xrightarrow{u} (p'_n, \varepsilon) \in \mathit{sruns}(\mathcal{A}). \tag{B.3.i}$$

with $u = k_1 v_1 \# k_2 v_2 \# \dots \# k_n v_n$ such that

$$\forall i : v_i \in \Sigma_{\text{pVal}} \cup \{a u \bar{a} \mid a \in \Sigma_c, u \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}})\}.$$

As p'_n is not a bin state, there exists another stacked run

$$(q_0, \varepsilon) \xrightarrow{t} (p'_n, \sigma) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}) \tag{B.3.ii}$$

with $q \in F$.

Let us decompose t in terms of its unmatched call symbols, *i. e.*,

$$t = t_1 \cdot a_1 \cdot t_2 \cdot a_2 \dots t_m \cdot a_m \cdot t_{m+1}$$

such that $m \geq 0$ and $t_i \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}}), a_i \in \Sigma_c$ for all i. It follows that in (B.3.ii), $|\sigma| = m$. Therefore, with $t = t'' \cdot t_{m+1}$ and since $\mathcal{A}$ is a 1-SEVPA, we get the stacked run

$$(q_0, \varepsilon) \xrightarrow{t''} (q_0, \sigma) \xrightarrow{t_{m+1}} (p'_n, \sigma) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}).$$

By (B.3.i), we can replace t_{m+1} by u showing that the word $t'' \cdot u \cdot t'$ belongs to $\mathcal{L}(\mathcal{A}) = \mathcal{L}_<(\mathcal{G})$. Moreover, as each

$$v_i \in \Sigma_{\text{pVal}} \cup \{a u \bar{a} \mid a \in \Sigma_c, u \in \mathrm{WM}(\widetilde{\Sigma}_{\text{JSON}})\},$$

it follows that $k_i \cdot v_i$ is a key-value pair for all i.

B.4. Complexity of the key graph

We recall that the algorithm for computing the key graph $\mathrm{G}_{\mathcal{A}}$ from the 1-SEVPA $\mathcal{A}$ requires three main steps:

1. compute the reachability relation $\mathrm{Reach}_{\mathcal{A}}$,
2. detect the bin state and remove it from $\mathcal{A}$ (it is unique, if it exists), and
3. compute the vertices and edges of $\mathrm{G}_{\mathcal{A}}$.

We proceed as follows and give the related time complexities.[8]

8: Recall that $\mathcal{A}$ is deterministic meaning that given a left-hand side of a transition, we have access in constant time to its right-hand side.

B.4.1. Computing the reachability relation

First, we enrich the reachability relation $\mathrm{Reach}_{\mathcal{A}}$ with words as follows. We define a map $\mathrm{Wit}^R_{\mathcal{A}} : \mathrm{Reach}_{\mathcal{A}} \to \Sigma^*$ such that for all $(q, q') \in \mathrm{Reach}_{\mathcal{A}}$, we have $\mathrm{Wit}^R_{\mathcal{A}}(q, q') = w$ with

$$(q, \varepsilon) \xrightarrow{w} (q', \varepsilon) \in \mathit{sruns}(\mathcal{A}).$$

The word w is a *witness* of the membership of (q, q') to $\mathsf{Reach}_{\mathcal{A}}$. We compute this map as follows by choosing the witnesses step by step while computing $\mathsf{Reach}_{\mathcal{A}}$.[9]

9: An algorithm computing $\mathsf{Reach}_{\mathcal{A}}$ is given in Section 7.4.1.

- Initially, for all $q \in Q$, we define $\mathrm{Wit}^R_{\mathcal{A}}(q, q) = \varepsilon$, and for each (q, q') such that $q \neq q'$ and there exists an internal transition $q \xrightarrow{a} q'$, we define $\mathrm{Wit}^R_{\mathcal{A}}(q, q') = a$.
- During the computation of $\mathsf{Reach}_{\mathcal{A}}$, there are two scenarios where a new element (q, q') is added to $\mathsf{Reach}_{\mathcal{A}}$
  1. There exists a call transition $q \xrightarrow{a/\gamma} p$ and a return transition $p' \xrightarrow{\bar{a}[\gamma]} q'$, with $(p, p') \in \mathsf{Reach}_{\mathcal{A}}$. In that case, let $\mathrm{Wit}^R_{\mathcal{A}}(p, p') = w$ the already known witness and define
  $$\mathrm{Wit}^R_{\mathcal{A}}(q, q') = a \cdot w \cdot \bar{a}.$$
  2. There exist $(q, p), (p, q') \in \mathsf{Reach}_{\mathcal{A}}$. In that case, we apply the transitive closure: let $\mathrm{Wit}^R_{\mathcal{A}}(q, p) = w$ and $\mathrm{Wit}^R_{\mathcal{A}}(p, q') = w'$, and define
  $$\mathrm{Wit}^R_{\mathcal{A}}(q, q') = w \cdot w'.$$

Lemma B.4.1. *Computing the relation* $\mathsf{Reach}_{\mathcal{A}}$ *enriched with the map* $\mathrm{Wit}^R_{\mathcal{A}}$ *is in time*
$$\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^5 + |\delta| \cdot \left|Q^{\mathcal{A}}\right|^4\right).$$

Proof. The relation $\mathsf{Reach}_{\mathcal{A}}$ with its witnesses are stored in a matrix of size $\left|Q^{\mathcal{A}}\right|^2$. The initialization is in $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^2 + |\delta_{int}|\right)$.

The main loop uses at most $\left|Q^{\mathcal{A}}\right|^2$ steps. One operation inside this loop is the transitive closure that can be computed in $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^3\right)$ with Warshall's algorithm. The other operation is in $\mathcal{O}\left(|\delta_c| \cdot \left|Q^{\mathcal{A}}\right|^2\right)$.

Therefore, the overall complexity of computing $\mathsf{Reach}_{\mathcal{A}}$ is in
$$\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^5 + |\delta| \cdot \left|Q^{\mathcal{A}}\right|^4\right).$$ □

B.4.2. Computing the bin state

Second, for detecting the bin state of $\mathcal{A}$, we define the following set $Z_{\mathcal{A}} \subseteq Q^{\mathcal{A}}$:

$$\begin{aligned} Z_{\mathcal{A}} = \{p \in Q^{\mathcal{A}} \mid \exists q \in F : (q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w} (q_0^{\mathcal{A}}, \sigma) \in \mathit{sruns}(\mathcal{A}) \\ \text{and } (p, \sigma) \xrightarrow{w'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})\}, \end{aligned}$$

enriched with the witness map $\mathrm{Wit}^{\perp}_{\mathcal{A}} : Z_{\mathcal{A}} \to (\Sigma^*)^2$ that assigns to each $p \in Z_{\mathcal{A}}$ a pair of words (w, w') as in the previous definition. In the definition of $Z_{\mathcal{A}}$, notice the same stack content σ in both configurations $(q_0^{\mathcal{A}}, \sigma)$ and (p, σ), and the presence of $q_0^{\mathcal{A}}$ in the first configuration.[10]

10: We can restrict to $q_0^{\mathcal{A}}$, as we work with 1-SEVPAs having all their call transitions going to $q_0^{\mathcal{A}}$.

From this set $Z_{\mathcal{A}}$, we easily derive the next lemma.

Lemma B.4.2. *For each state $p \in Q^{\mathcal{A}}$, p is not a bin state if and only if $p \in Z_{\mathcal{A}}$.*

Proof. Assume p is not a bin state, *i.e.*, by definition, there exists a stacked run traversing p:

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{t} (p, \sigma) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})$$

with $q \in F$. By repeating the argument given in the proof of Lemma 7.4.2 (see (B.3.ii)), we get that this path decomposes as

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{t''} (q_0^{\mathcal{A}}, \sigma) \xrightarrow{t_{m+1}} (p, \sigma) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})$$

with $t = t'' \cdot t_{m+1}$ such that t'' contains m unmatched call symbols and $t_{m+1} \in \mathrm{WM}(\widetilde{\Sigma}_{\mathrm{JSON}})$.[11] This shows that $p \in Z_{\mathcal{A}}$.
Now, assume $p \in Z_{\mathcal{A}}$, *i.e.*, there exists two stacked runs

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w} (q_0^{\mathcal{A}}, \sigma) \in \mathit{sruns}(\mathcal{A})$$

and

$$(p, \sigma) \xrightarrow{w'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A})$$

with $q \in F$. If we prove that $(q_0^{\mathcal{A}}, p) \in \mathrm{Reach}_{\mathcal{A}}$, we are done. As $\mathcal{A}$ is a minimal 1-SEVPA,[12] there exists some stacked run

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{t} (p, \sigma') \in \mathit{sruns}(\mathcal{A}).$$

As done above with (B.3.ii), we get that this stacked run decomposes as

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{t''} (q_0^{\mathcal{A}}, \sigma') \xrightarrow{t_{m+1}} (p, \sigma') \in \mathit{sruns}(\mathcal{A}),$$

with t'' a word with m unmatched call symbols, and t_{m+1} a well-matched word. As $(q_0^{\mathcal{A}}, \sigma') \xrightarrow{t_{m+1}} (p, \sigma')$ is a stacked run of $\mathcal{A}$, it follows that $(q_0^{\mathcal{A}}, p) \in \mathrm{Reach}_{\mathcal{A}}$.
Observe that, since t_{m+1} is a well-matched word, it is possible to read it from any configuration $(q_0^{\mathcal{A}}, \cdot)$. Hence, we have the stacked run

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{w} (q_0^{\mathcal{A}}, \sigma) \xrightarrow{t_{m+1}} (p, \sigma) \xrightarrow{w'} (q, \varepsilon),$$

showing that p is not a bin. □

11: Observe that t_{m+1} may contain call and return symbols, as long as the balance of t_{m+1} is null.

12: So, every state of $\mathcal{A}$ is reachable from $q_0^{\mathcal{A}}$ by reading a well-chosen (potentially unbalanced) word.

We compute $Z_{\mathcal{A}}$ and $\mathrm{Wit}_{\mathcal{A}}^{\perp}$ as follows:

- Initially, add q to $Z_{\mathcal{A}}$ for all $q \in F$, and assign the witness $\mathrm{Wit}_{\mathcal{A}}^{\perp}(q) = (\varepsilon, \varepsilon)$ to q.
- Then, repeat until $Z_{\mathcal{A}}$ stabilizes:

Direct reachability If we have $(p', p) \in \mathrm{Reach}_{\mathcal{A}}$ with $p \in Z_{\mathcal{A}}$, $\mathrm{Wit}_{\mathcal{A}}^{\perp}(p) = (w, w')$, and $\mathrm{Wit}_{\mathcal{A}}^{R}(p', p) = t'$, then add[13] p' to $Z_{\mathcal{A}}$ with

$$\mathrm{Wit}_{\mathcal{A}}^{\perp}(p') = (w, t' \cdot w').$$

13: If p' already belongs to $Z_{\mathcal{A}}$, we do nothing.

Via call and return If we have a call transition $r \xrightarrow{a/\gamma} q_0^{\mathcal{A}}$ and a return transition $p' \xrightarrow{\bar{a}[\gamma]} p$ such that $p \in Z_{\mathcal{A}}$ and $(q_0^{\mathcal{A}}, r) \in \mathrm{Reach}_{\mathcal{A}}$ with $\mathrm{Wit}_{\mathcal{A}}^{\perp}(p) = (w, w')$ and $\mathrm{Wit}_{\mathcal{A}}^{R}(q_0^{\mathcal{A}}, r) = t$ then add p' to $Z_{\mathcal{A}}$ with

$$\mathrm{Wit}_{\mathcal{A}}^{\perp}(p') = (w \cdot t \cdot a, \bar{a} \cdot w').$$

This algorithm is correct as shown by the next lemma.

Lemma B.4.3. *Let Z computed by the algorithm above. Then, $Z = Z_{\mathcal{A}}$.*

Proof. It is easy to see that $Z \subseteq Z_{\mathcal{A}}$. Let us prove that $Z_{\mathcal{A}} \subseteq Z$. Let $p \in Z_{\mathcal{A}}$. That is, we have $w, w' \in \Sigma^*, \sigma \in \Gamma^*$ and $q \in F$ such that

$$\begin{aligned} (q_0^{\mathcal{A}}, \varepsilon) &\xrightarrow{w} (q_0^{\mathcal{A}}, \sigma) \in \mathit{sruns}(\mathcal{A}) \\ (p, \sigma) &\xrightarrow{w'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}). \end{aligned} \tag{B.4.i}$$

We prove by induction over $\beta(w) = -\beta(w')$ that $p \in Z$.

Base case. $\beta(w) = 0$, *i.e.*, $\sigma = \varepsilon$. Thus, we can focus on the witnesses (ε, w') instead of (w, w'), as

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{\varepsilon} (q_0^{\mathcal{A}}, \varepsilon) \in \mathit{sruns}(\mathcal{A}).$$

Moreover, we have $w' \in \mathrm{WM}(\widetilde{\Sigma})$ and $(p, q) \in \mathrm{Reach}_{\mathcal{A}}$. By the initialization, $q \in Z$, and by the item *Direct reachability*, $p \in Z$.

Induction step. $\beta(w) > 0$. In that case, we can decompose $w = tw_1aw_2$, with $t \in (\mathrm{WM}(\widetilde{\Sigma}) \cdot \Sigma_c)^*, a \in \Sigma_c, w_1, w_2 \in \mathrm{WM}(\widetilde{\Sigma})$, and $w' = w''\bar{a}t'$, with $w'' \in \mathrm{WM}(\widetilde{\Sigma}), \bar{a} \in \Sigma_r$. We can thus decompose the stack runs of (B.4.i) into

$$\begin{aligned} (q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{t} (q_0^{\mathcal{A}}, \sigma_1) \xrightarrow{w_1} (r, \sigma_1) \xrightarrow{a} (q_0^{\mathcal{A}}, \sigma) \\ \xrightarrow{w_2} (q_0^{\mathcal{A}}, \sigma) \in \mathit{sruns}(\mathcal{A}) \end{aligned}$$

and

$$(p, \sigma) \xrightarrow{w''} (s, \sigma) \xrightarrow{\bar{a}} (s', \sigma_1) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}).$$

Therefore, we can focus on the witnesses (tw_1a, w') instead of (w, w'). Since

$$(q_0^{\mathcal{A}}, \varepsilon) \xrightarrow{t} (q_0^{\mathcal{A}}, \sigma_1) \in \mathit{sruns}(\mathcal{A})$$

and

$$(s', \sigma_1) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{A}),$$

it holds that $s' \in Z_{\mathcal{A}}$ and, by induction, $s' \in Z$. By the item *Via call and return*, $s \in Z$. Finally, by the step *Direct reachability*, $p \in Z$.
In conclusion, we have $p \in Z_{\mathcal{A}} \Rightarrow p \in Z$. As said above, it is easy to see that $Z \subseteq Z_{\mathcal{A}}$, *i.e.*, that $p \in Z \Rightarrow p \in Z_{\mathcal{A}}$. Hence, we have the claimed equivalence. □

Lemma B.4.4. *Detecting and removing the bin state of $\mathcal{A}$, if it exists, is in time*

$$\mathcal{O}\left(|\delta| \cdot |Q^{\mathcal{A}}|^4\right).$$

Proof. Let us first show that computing the set $Z_{\mathcal{A}}$ is in

$$\mathcal{O}\left(|\delta| \cdot |Q^{\mathcal{A}}|^4\right).$$

The initialization is in $\mathcal{O}(|Q^{\mathcal{A}}|)$. The main loop uses at most $|Q^{\mathcal{A}}|$ steps until $Z_{\mathcal{A}}$ stabilizes. Its body is in $\mathcal{O}\left(|Q^{\mathcal{A}}|^3 + |\delta_r| \cdot |Q^{\mathcal{A}}|^3\right)$. Second, detecting the bin state of $\mathcal{A}$ can be done in $\mathcal{O}\left(|Q^{\mathcal{A}}|^2\right)$ by iterating over all values of $Z_{\mathcal{A}}$ (thanks to Lemma B.4.2).
Finally, removing this bin state from $\mathcal{A}$ and its related transitions is in $\mathcal{O}(|Q^{\mathcal{A}}| + |\delta|)$. Therefore we get the complexity announced in the lemma. □

Lemma B.4.2. For each state $p \in Q^{\mathcal{A}}$, p is not a bin state if and only if $p \in Z_{\mathcal{A}}$.

B.4.3. Computing the key graph

Third, the vertices and the edges $\mathrm{G}_{\mathcal{A}}$ are computed as follows: (p, k, p') is a vertex in $\mathrm{G}_{\mathcal{A}}$ if there exist an internal transition $p \xrightarrow{k} q$ with $k \in \Sigma_{\text{key}}$ and

- an internal transition $q \xrightarrow{a} p'$ with $a \in \Sigma_{\text{pVal}}$, or
- a call transition $q \xrightarrow{a/\gamma} r$ and a return transition $r' \xrightarrow{\bar{a}[\gamma]} p'$, with $(r, r') \in \mathrm{Reach}_{\mathcal{A}}$.

and $((p_1, k_1, p_1'), (p_2, k_2, p_2'))$ is an edge in $\mathrm{G}_{\mathcal{A}}$ if there exists an internal transition $p_1' \xrightarrow{\#} p_2$.

Lemma B.4.5. *Constructing the key graph $\mathrm{G}_{\mathcal{A}}$ is in*

$$\mathcal{O}\left(|\delta|^2 + |\delta| \cdot |Q^{\mathcal{A}}|^2 + |Q^{\mathcal{A}}|^4 \cdot |\Sigma_{\text{key}}|^2\right).$$

Proof. Constructing the vertices of $\mathrm{G}_{\mathcal{A}}$ is in

$$\mathcal{O}\left(|\delta_{int}|^2 + |\delta_{int}| \cdot |Q^{\mathcal{A}}|^2\right).$$

Constructing its edges is in

$$\mathcal{O}\left(|Q^{\mathcal{A}}|^4 \cdot |\Sigma_{\text{key}}|^2\right)$$

by Lemma 7.4.5. We thus immediately obtain the lemma. □

Lemma 7.4.5. The key graph $\mathrm{G}_{\mathcal{A}}$ has $\mathcal{O}\left(|Q^{\mathcal{A}}|^2 \cdot |\Sigma_{\text{key}}|\right)$ vertices. Moreover, visiting all vertices along all the paths of $\mathrm{G}_{\mathcal{A}}$ that start from a vertex (p, k, p') such that $p = q_0^{\mathcal{A}}$ is in $\mathcal{O}\left(|Q^{\mathcal{A}} \times \Sigma_{\text{key}}|^{|\Sigma_{\text{key}}|}\right)$.

The complexity announced in Proposition 7.4.6 (repeated just after) follows from the previous Lemmas B.4.1, B.4.4 and B.4.5

Proposition 7.4.6. *Computing the key graph* $\mathrm{G}_{\mathcal{A}}$ *is in time*

$$\mathcal{O}\left(|\delta|^2 + |\delta| \cdot \left|Q^{\mathcal{A}}\right|^4 + \left|Q^{\mathcal{A}}\right|^5 + \left|Q^{\mathcal{A}}\right|^4 \cdot \left|\Sigma_{\text{key}}\right|^2\right).$$

B.5. Proof of Theorem 7.4.7

Theorem 7.4.7. *Let S be a JSON schema defined by a closed extended CFG $\mathcal{G}$ and $\mathcal{A}$ be a 1-SEVPA $\mathcal{A}$ accepting $\mathcal{L}_<(\mathcal{G})$. Then, checking whether a JSON document J with depth $depth(J)$ satisfies the schema S*

- *is in time*

$$\mathcal{O}\left(|J| \cdot \left(|Q|^4 + |Q|^{\left|\Sigma_{\text{key}}\right|} \cdot \left|\Sigma_{\text{key}}\right|^{\left|\Sigma_{\text{key}}\right|+1}\right)\right),$$

- *and uses an amount of memory in*

$$\mathcal{O}\left(|\delta| + |\mathcal{A}|^2 \cdot \left|\Sigma_{\text{key}}\right| + depth(J) \cdot \left(|\mathcal{A}|^2 + \left|\Sigma_{\text{key}}\right|\right)\right).$$

We start with the time complexity of Algorithm 7.1. Before doing so, let us mention that in addition to the key graph $\mathrm{G}_{\mathcal{A}}$, for each key $k \in \Sigma_{\text{key}}$, we have a list, denoted by $List_k$, in which all the vertices in $\mathrm{G}_{\mathcal{A}}$ of the form (p, k, p') are stored. Those lists are useful to compute the set *Bad* (see Lines 20 and 30 of Algorithm 7.1).

Let us also comment on how the set $\mathrm{Valid}(K, Bad)$ is computed in Line 21. Recall that $\mathrm{G}_{\mathcal{A}}$ has a finite number of paths (see Corollary 7.4.4) and that each element (r, r') of $\mathrm{Valid}(K, Bad)$ is such that $r = q_0^{\mathcal{A}}$. By a recursive algorithm, we visit each path of $\mathrm{G}_{\mathcal{A}}$ starting with any vertex of the form (p, k, p') with $p = q_0^{\mathcal{A}}$. We stop visiting such a path as soon as we visit a vertex containing a key $k \notin K$ or belonging to *Bad*. During the visit of the current path, we collect the keys appearing in its vertices in a set K'. When the path reaches some vertex (r, k, r') with $K' = K$, then we add $(q_0^{\mathcal{A}}, r')$ to $\mathrm{Valid}(K, Bad)$. Hence, computing $\mathrm{Valid}(K, Bad)$ is in

Corollary 7.4.4. In the key graph $\mathrm{G}_{\mathcal{A}}$, there is no path $((p_1, k_1, p'_1)(p_2, k_2, p'_2) \dots (p_n, k_n, p'_n))$ with $p_1 = q_0^{\mathcal{A}}$ such that $k_i = k_j$ for some $i \neq j$.

$$\mathcal{O}\left(\left|\Sigma_{\text{key}}\right| \cdot \left|Q^{\mathcal{A}} \times \Sigma_{\text{key}}\right|^{\left|\Sigma_{\text{key}}\right|}\right) = \mathcal{O}\left(|Q|^{\left|\Sigma_{\text{key}}\right|} \cdot \left|\Sigma_{\text{key}}\right|^{\left|\Sigma_{\text{key}}\right|+1}\right)$$

by Lemma 7.4.5 and because checking whether K' and K are equal is in $\mathcal{O}\left(\left|\Sigma_{\text{key}}\right|\right)$.

Lemma 7.4.5. The key graph $\mathrm{G}_{\mathcal{A}}$ has $\mathcal{O}\left(\left|Q^{\mathcal{A}}\right|^2 \cdot \left|\Sigma_{\text{key}}\right|\right)$ vertices. Moreover, visiting all vertices along all the paths of $\mathrm{G}_{\mathcal{A}}$ that start from a vertex (p, k, p') such that $p = q_0^{\mathcal{A}}$ is in $\mathcal{O}\left(\left|Q^{\mathcal{A}} \times \Sigma_{\text{key}}\right|^{\left|\Sigma_{\text{key}}\right|}\right)$.

Let us now consider each case of Algorithm 7.1 and study its complexity (when **yes** is returned). Recall that $\mathcal{A}$ is deterministic meaning that given a left-hand side of a transition, we have access in constant time to its right-hand side. Notice that at several places, the current set $R \subseteq Q^2$ is updated as $\mathbb{I}_P$ for some subset $P \subseteq Q$, which can be done in $\mathcal{O}\left(|Q|^2\right)$. The different cases are the following ones:

- The cases $a = \sqsubset$ (Line 5) and $a = \prec$ (Line 6) are in $\mathcal{O}\left(|Q|^2\right)$.

- The case $a = \sqsupset$ (Line 12) is in $\mathcal{O}(|R| \cdot |R'|)$ for computing the updated set

$$R_{Upd} = \{(p, q) \mid \exists (p, p') \in R', p' \xrightarrow{\sqsubset/\gamma} q_0^{\mathcal{A}}, (q_0^{\mathcal{A}}, r) \in R, r \xrightarrow{\sqsupset[\gamma]} q\}.$$

Indeed we have access in constant time to the call transition $p' \xrightarrow{\sqsubset/\gamma} q_0^{\mathcal{A}}$ and the return transition $r \xrightarrow{\sqsupset[\gamma]} q$ when computing R_{Upd}. Therefore, this case is in $\mathcal{O}\left(|Q|^4\right)$.
- The case $a \in \Sigma_{int}$ with $\sqsubset$ appearing on top of Stk or $a \neq \#$ if $\prec$ appears on top of Stk (Line 24) is in $\mathcal{O}\left(|Q|^2\right)$.
- The case $a = \#$ (Line 26) is in $\mathcal{O}\left(|Q|^2\right)$. Indeed, finding the vertices (p, k, p') that have to be added to *Bad* can be done in $\mathcal{O}\left(|Q|^2\right)$ by traversing the list $List_k$.
- The case $a = \succ$ (Lines 15 and 18) has some similarities with the case $a = \sqsupset$. In case $(R', \prec)$ is popped, then this step is in $\mathcal{O}\left(|Q|^2\right)$ as R does not appear in the computation of R_{Upd}. In the other case, we need to compute the set $\text{Valid}(K, Bad) \subseteq Q^2$. This step is thus in

$$\mathcal{O}\left(|Q|^4 + |Q|^{|\Sigma_{\text{key}}|} \cdot |\Sigma_{\text{key}}|^{|\Sigma_{\text{key}}|+1}\right).$$

Therefore, the overall time complexity of Algorithm 7.1 is in

$$\mathcal{O}\left(|J| \cdot \left(|Q|^4 + |Q|^{|\Sigma_{\text{key}}|} \cdot |\Sigma_{\text{key}}|^{|\Sigma_{\text{key}}|+1}\right)\right).$$

We now proceed with the memory complexity. Algorithm 7.1 uses auxiliary memory to store:

- the given 1-SEVPA $\mathcal{A}$,
- its key graph $G_{\mathcal{A}}$ with the lists $List_k$, $k \in \Sigma_{\text{key}}$, as introduced above,
- the current set R, the current symbol a and the symbol b following a,
- the stack Stk whose elements are of the form either $(R', \sqsubset)$, or $(R', \prec)$, or $(R', \prec, K, k, Bad)$, and
- the set $\text{Valid}(K, Bad)$.

Let us denote by $|\mathcal{A}|$ (resp. $|\delta|$) the number of states (resp. of transitions) of the 1-SEVPA $\mathcal{A}$. By Lemma 7.4.5, the key graph has $\mathcal{O}\left(|\mathcal{A}|^2 \cdot |\Sigma_{\text{key}}|\right)$ vertices.[14] The same $\mathcal{O}\left(|\mathcal{A}|^2 \cdot |\Sigma_{\text{key}}|\right)$ holds for the lists $List_k$, $k \in \Sigma_{\text{key}}$, as they are together composed of the vertices of $G_{\mathcal{A}}$.

The sizes of R and $\text{Valid}(K, Bad)$ are in $\mathcal{O}\left(|\mathcal{A}|^2\right)$ as they are subsets of Q^2. The biggest elements in the stack Stk are of the form $(R', \prec, K, k, Bad)$ with $K \subseteq \Sigma_{\text{key}}$ and *Bad* containing some vertices of $G_{\mathcal{A}}$, thus with a size in $\mathcal{O}\left(|\mathcal{A}|^2 + |\Sigma_{\text{key}}|\right)$.

The number of elements stored in the stack Stk is bounded by the depth $depth(J)$ of the JSON document J.

All in all, the memory used by Algorithm 7.1 is in

$$\mathcal{O}\left(|\delta| + |\mathcal{A}|^2 \cdot |\Sigma_{\text{key}}| + depth(J) \cdot (|\mathcal{A}|^2 + |\Sigma_{\text{key}}|)\right).$$

Lemma 7.4.5. The key graph $G_{\mathcal{A}}$ has $\mathcal{O}\left(|Q^{\mathcal{A}}|^2 \cdot |\Sigma_{\text{key}}|\right)$ vertices. Moreover, visiting all vertices along all the paths of $G_{\mathcal{A}}$ that start from a vertex (p, k, p') such that $p = q_0^{\mathcal{A}}$ is in $\mathcal{O}\left(|Q^{\mathcal{A}} \times \Sigma_{\text{key}}|^{|\Sigma_{\text{key}}|}\right)$.

14: It is not necessary to count its transitions as it is acyclic, by Corollary 7.4.4.

Corollary 7.4.4. In the key graph $G_{\mathcal{A}}$, there is no path $((p_1, k_1, p_1')(p_2, k_2, p_2') \dots (p_n, k_n, p_n'))$ with $p_1 = q_0^{\mathcal{A}}$ such that $k_i = k_j$ for some $i \neq j$.

B.6. Generating a counterexample from the key graph

The goal of this section is to prove the following lemma, allowing us to perform the last check in an equivalence query (see Section 7.5.2).

Lemma B.6.1. *et $\mathcal{H}$ be an automaton constructed by the learner. If the key graph $\mathrm{G}_{\mathcal{H}}$ contains a path $((p_1, k_1, p'_1) \dots (p_n, k_n, p'_n))$ with $p_1 = q_0$ such that $k_i = k_j$ for some $i \neq j$, then one can construct a word accepted by $\mathcal{H}$ that is not a valid JSON document.*

We proceed exactly as in the proof of Lemma 7.4.2. In the key graph $\mathrm{G}_{\mathcal{H}}$ of $\mathcal{H}$, let

$$((p_1, k_1, p'_1)(p_2, k_2, p'_2) \dots (p_n, k_n, p'_n)) \tag{B.6.i}$$

be a path with $p_1 = q_0^{\mathcal{H}}$ such that $k_i = k_j$ for some $i \neq j$. Then, in the proof of Lemma 7.4.2, we proved that there exists a stacked run

$$(q_0, \varepsilon) \xrightarrow{t''} (q_0, \sigma) \xrightarrow{u} (p'_n, \sigma) \xrightarrow{t'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{H}) \tag{B.6.ii}$$

with $q \in F^{\mathcal{H}}$ such that $u = k_1 v_1 \# \dots \# k_n v_n$ is part of an object. his shows that the word $t'' \cdot u \cdot t'$ is accepted by $\mathcal{H}$ and is not a valid document by the presence of the repeated keys k_i, k_j. Let us explain how to *construct* this word $t'' \cdot u \cdot t'$.

For this purpose, we are going to use the witnesses introduced in Section B.4. First, thanks to the map $\mathrm{Wit}_{\mathcal{H}}$ associated with $\mathrm{Reach}_{\mathcal{H}}$ and computed in Section B.4, we can similarly enrich the vertices of $\mathrm{G}_{\mathcal{H}}$ with witnesses: we assign a key-value pair $k \cdot v = \mathrm{Wit}_{\mathcal{H}}(p, k, p')$ to each vertex (p, k, p') of $\mathrm{G}_{\mathcal{H}}$ such that

$$(q, \varepsilon) \xrightarrow{k \cdot v} (q', \varepsilon) \in \mathit{sruns}(\mathcal{H}).$$

Therefore, from a path in $\mathrm{G}_{\mathcal{H}}$ like (B.6.i), we derive the witness $u = k_1 v_1 \# \cdots \# k_n v_n$ such that

$$(q_0, \varepsilon) \xrightarrow{u} (p'_n, \varepsilon) \in \mathit{sruns}(\mathcal{H}).$$

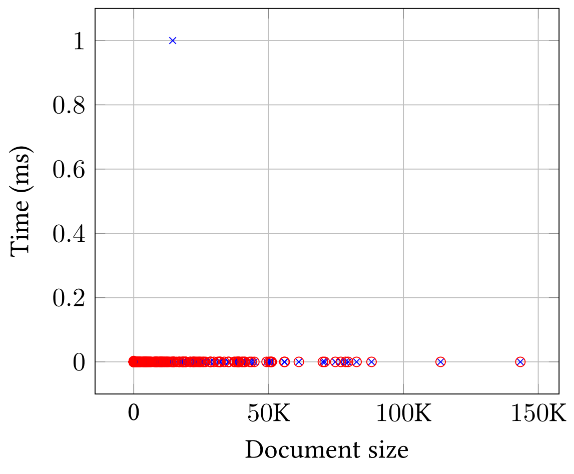
It remains to extend this witness u into a witness $t'' \cdot u \cdot t'$ of a stacked run like in (B.6.ii). This is possible by noticing that $p'_n \in Z_{\mathcal{A}}$. Therefore with the computed witness $\mathrm{Wit}'_{\mathcal{H}}(q_0, p'_n) = (w, w')$ and the stacked run

$$(q_0, \varepsilon) \xrightarrow{u} (p'_n, \varepsilon) \in \mathit{sruns}(\mathcal{H}),$$

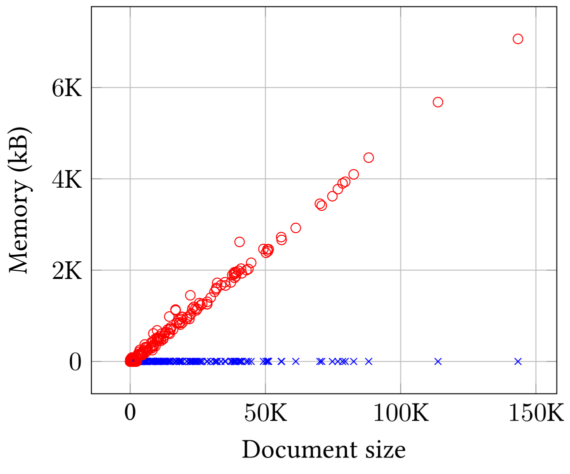
we get the stacked run

$$(q_0, \varepsilon) \xrightarrow{w} (q_0, \sigma) \xrightarrow{u} (p'_n, \sigma) \xrightarrow{w'} (q, \varepsilon) \in \mathit{sruns}(\mathcal{H})$$

for some $q \in F^{\mathcal{H}}$. The witness $w \cdot u \cdot w'$ of this path is the required counterexample.

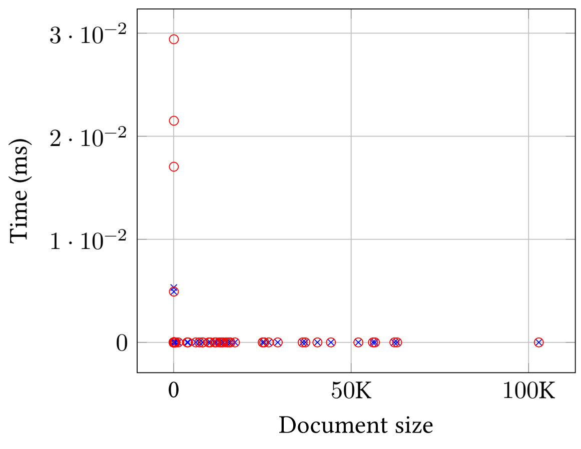


(a) Time.

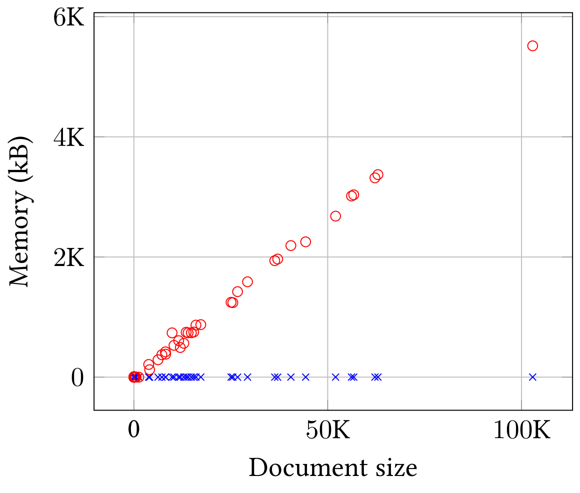


(b) Memory.

Figure B.1: Results of validation benchmarks for the recursive list schema.



(a) Time.

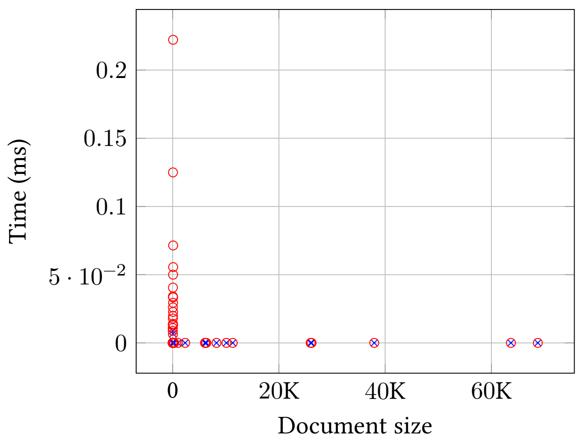


(b) Memory.

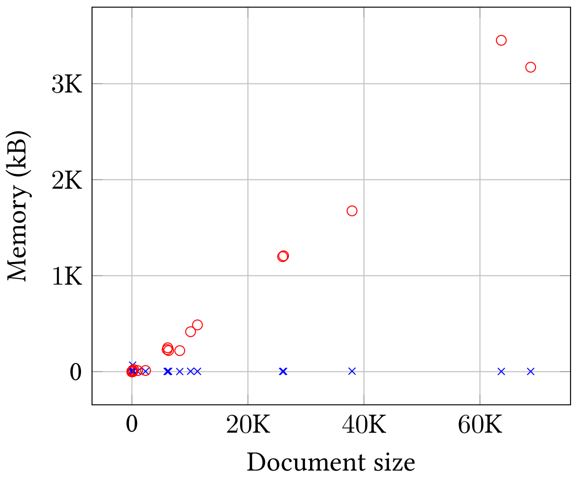
Figure B.2: Results of validation benchmarks for the schema iterating over the types of values.

B.7. Validation results for the first three schemas

Results for the comparison of the classical and our new validation algorithms on the first three schemas of Section 7.5 are given in Figures B.1 to B.3. Recall that blue crosses give the values for our algorithm, while the red circles stand for the classical algorithm.



(a) Time.



(b) Memory.

Figure B.3: Results of validation benchmarks for the snippet configuration of *Visual Studio Code*.

Part IV.

Mealy Machines with Timers

Introduction and Timed Mealy machines 8.

In the third part, Mealy Machines with Timers, we extend Mealy machines[1] with resources that can measure the time that elapsed since a previous event. We present two different types of resources:

1: See Section 2.2.2.

- The first ones are *clocks*. Each transition of a *timed Mealy machine* can reset a set of clocks (set their values to zero). Then, as time goes on, the values of the clocks increase. Furthermore, each transition has a *guard*, which is a condition over the clocks, *i.e.*, the transition can be triggered only when the current values of the clocks satisfy the guard.
  Timed Mealy machines are the focus of this chapter.
- The second ones are *timers*. A transition of a *Mealy machine with timers* can set the value of a timer to a natural constant. Then, as time goes on, the values of the timers decrease. When a timer reaches zero, a special symbol, called a *timeout*, occurs.
  This model is introduced in Chapter 9, while Chapter 10 provides a learning algorithm for Mealy machines with timers.

Chapter contents

8.1. Introduction

Many systems used in practice have some form of timing constraints. For instance, some switches allow to control the intensity of the light depending on how long the switch is pressed. If one were to model such a switch, the considered automaton ought to be able to measure time. Alur and Dill [AD94] introduced *timed automata*, which are finite state automata equipped with real-valued clock variables that measure the time between state transitions. These clock variables all increase at the same rate when time elapses, and can be reset and used in guards along transitions. Timed automata have become a framework of choice for modeling and analysis of real-time systems, equipped with a rich theory, supported by powerful tools (such as UPPAAL [Beh+06] and TChecker[2]), and with numerous applications [Bou+18; Cla+18; BK08; BY03].

[AD94]: Alur et al. (1994), "A Theory of Timed Automata"

[Beh+06]: Behrmann et al. (2006), "UPPAAL 4.0"

2: `https://github.com/ticktac-project/tchecker`

[Bou+18]: Bouyer et al. (2018), "Model Checking Real-Time Systems"

[Cla+18]: Clarke et al. (2018), *Handbook of Model Checking*

[BK08]: Baier et al. (2008), *Principles of model checking*

[BY03]: Bengtsson et al. (2003), "Timed Automata: Semantics, Algorithms and Tools"

In this chapter, we present *timed Mealy machines*, which are a straightforward adaptation of timed automata where each transition outputs a symbol. Beside the semantics of timed Mealy machines, we recall here two results for timed systems, which will be useful for the next chapters:

- In Section 8.3, we study the problem of deciding whether a given state can be reached by some (unknown) timed run. It is known to be PSPACE-complete [AL02]. While the hardness comes from the reduction from the acceptance problem for *linear bounded Turing machines*, the membership is obtained by abstracting the timed behavior of the machine. More specifically, we group together valuations of the clocks that share common properties (called *regions*), which yields a finite-state machine of exponential size. As the region machine can be computed on the fly, we obtain the PSPACE membership.
- Since the number of states obtained by the above approach is extremely large, we introduce a more efficient representation of the state space, using the notions of *zones*, in Section 8.4.

[AL02]: Aceto et al. (2002), "Is your model checker on time? On the complexity of model checking for timed modal logics"

Furthermore, Section 8.5 lists various other interesting results. We refer to [Cla+18; BK08] for a more thorough introduction to timed automata.

For information, there exist active learning algorithms for timed automata, that each consider various restrictions:

- when a single clock can be used [An+20],
- when each transition can reset at most one clock [An+21],
- when a clock is associated to each input symbol and measures the time that elapsed since the last time that symbol was read [GJL10; HJM20],
- when the automaton is assumed to be deterministic but can use an arbitrary number of clocks (without any further restriction) [Wag23],
- and many more [APT20; TZA24; VWW07].

Furthermore, some works apply automata learning to verify systems, such as [San23].

[An+20]: An et al. (2020), "Learning One-Clock Timed Automata"

[An+21]: An et al. (2021), "Learning real-time automata"

[GJL10]: Grinchtein et al. (2010), "Learning of event-recording automata"

[HJM20]: Henry et al. (2020), "Active Learning of Timed Automata with Unobservable Resets"

[Wag23]: Waga (2023), "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization"

[APT20]: Aichernig et al. (2020), "From Passive to Active: Learning Timed Automata Efficiently"

[TZA24]: Teng et al. (2024), "Learning Deterministic Multi-Clock Timed Automata"

[VWW07]: Verwer et al. (2007), "An algorithm for learning real-time automata"

[San23]: Sankur (2023), "Timed Automata Verification and Synthesis via Finite Automata Learning"

8.2. Timed Mealy machines

Let us define timed Mealy machines and their semantics. Our presentation is inspired by [Cla+18, Chapter 29]. As said above, such a machine has a set of *clocks* that can have any non-negative real value (*i.e.*, is in $\mathbb{R}^{\geq 0}$). We first define the syntax of the model before giving its semantics.

Let C be a set of clocks. A *clock constraint* over C is a formula described by the following grammar:

$$\phi ::= x < c \mid x \leq c \mid c < x \mid c \leq x \mid \phi_1 \wedge \phi_2 \mid \top$$

with $x \in C$, $c \in \mathbb{N}$, and $\top$ denotes the constraint that is always satisfied. We denote by $\Phi(C)$ the set of all constraints following that grammar. We write $c_1 \leq x \leq c_2$ as a shortcut for $c_1 \leq x \wedge x \leq c_2$ for instance, and $x = c$ for $c \leq x \leq c$.

We sometimes make use of *diagonal clock constraints*, which additionally allow constraints of the form $x - y < c$ and $x - y \leq c$. We denote by $\Phi_D(C)$ for the union of $\Phi(C)$ and the set of all diagonal constraints.

A timed Mealy machine is then a Mealy machine augmented with a set of clocks C. Each transition possesses a *guard* (*i.e.*, a constraint in $\Phi(C)$) and

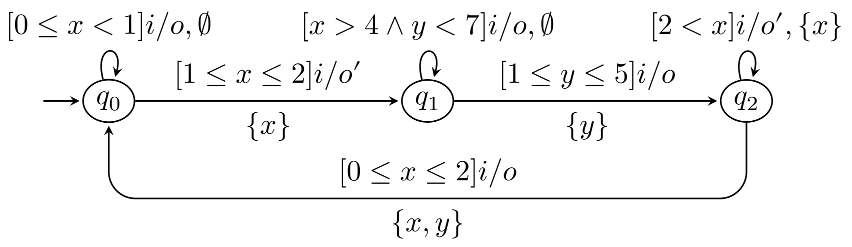


Figure 8.1: A sample timed Mealy machine such that $\mathit{Inv}(q_0) = 0 \leq x \leq 2$, $\mathit{Inv}(q_1) = 0 \leq x \leq 3 \wedge 0 \leq y \leq 5$, and $\mathit{Inv}(q_2) = 0 \leq y \leq 1$.

can *reset* a subset of C. Furthermore, a constraint over C, called an *invariant*, is given for each state. This will be used in the semantics to restrict how much time can elapse without triggering a transition, for instance. We focus here on *non-deterministic* machines.

Definition 8.2.1 (Timed Mealy machine)**.** A *timed Mealy machine* (*TMM*, for short) is a tuple $\mathcal{M} = (I, O, C, Q, q_0, \mathit{Inv}, \delta)$ where:

- Q and q_0 are the set of states and the initial state,
- $\mathit{Inv} : Q \to \Phi(C)$ is a function that maps an *invariant* to each state, and
- $\delta : (Q \times \Phi(C) \times I) \times (Q \times 2^C \times O)$ is the transition relation. As usual, we write $\delta(q, g, i)$ for the set of triplets (p, r, o) such that $((q, g, i), (p, r, o)) \in \delta$ and $q \xrightarrow[r]{[g]i/o} p$ when $(p, r, o) \in \delta(q, g, i)$.

As usual, we add, when needed, a superscript to indicate which TMM is considered, *e.g.*, $Q^{\mathcal{M}}, q_0^{\mathcal{M}}$, etc. Missing symbols in $q \xrightarrow[r]{[g]i/o} q'$ are quantified existentially, *e.g.*, $q \xrightarrow[r]{[g]i/o}$ means there exists q' such that $q \xrightarrow[r]{[g]i/o} q'$, and $q \xrightarrow{i}$ means there exist o, g, and u such that $q \xrightarrow[u]{[g]i/o}$.

A *run* π of $\mathcal{M}$ either consists of a single state p_0 or of a nonempty sequence of transitions, *i.e.*,

$$\pi = p_0 \xrightarrow[r_1]{[g_1]i_1/o_1} p_1 \xrightarrow[r_2]{[g_1]i_2/o_2} \cdots \xrightarrow[r_n]{[g_1]i_n/o_n} p_n.$$

We denote by $\mathit{runs}(\mathcal{M})$ the set of runs of $\mathcal{M}$. We often write $q \xrightarrow{[g]i} \in \mathit{runs}(\mathcal{M})$ to highlight that $\delta(q, g, i)$ is defined.

Example 8.2.2. Let $\mathcal{M}$ be the TMM of Figure 8.1 with $I = \{i\}$, $O = \{o, o'\}$, and $C = \{x, y\}$. Guards and resets are given along each transition. For instance, the transition from q_0 to q_1 requires that x is between 1 and 2, and resets x.
A sample run is

$$\pi = q_0 \xrightarrow[\emptyset]{[0 \leq x < 1]i/o} q_0 \xrightarrow[\{x\}]{[1 \leq x \leq 2]i/o'} q_1 \xrightarrow[\{y\}]{[1 \leq y \leq 5]i/o} q_2 \xrightarrow[\{x,y\}]{[0 \leq x \leq 2]i/o} q_0.$$

8.2.1. Semantics

Let us now define the timed semantics of a TMM, via an infinite-state labeled transition system describing all possible configurations and transitions between them.

A *valuation* is a (total) function $\kappa : C \to \mathbb{R}^{\geq 0}$ that assigns nonnegative real numbers to clocks. A *configuration* of $\mathcal{M}$ is a pair (q, κ) where $q \in Q$ and κ a valuation. The *initial configuration* is the pair (q_0, κ_0) where $\kappa_0(x) = 0$ for every $x \in C$.[3] When we let $d \in \mathbb{R}^{\geq 0}$ units of time elapse in a valuation κ, we write $\kappa + d$ for the resulting valuation that satisfies $(\kappa + d)(x) = \kappa(x) + d$ for all $x \in C$.

3: That is, we start with every clock already at zero.

We now define the conditions that must be satisfied for a transition from (q, κ) to (q', κ') to exist. We define two types of transitions: some wait a given delay in a state, while the others follow the transitions defined in δ. In every case, κ (resp. κ') must satisfy the invariant of q (resp. q'). This ensures that we only consider valuations that make sense with regards to the defined constraints.

Definition 8.2.3 (Timed run)**.** We define the transitions between configurations $(q, \kappa), (q', \kappa')$ as follows.

- $(q, \kappa) \xrightarrow{d} (q, \kappa')$ is a *delay transition*, if
  - κ satisfies $\mathit{Inv}(q)$,
  - $d \in \mathbb{R}^{\geq 0}$, and
  - $\kappa' = (\kappa + d)$ satisfies $\mathit{Inv}(q)$.
- $(q, \kappa) \xrightarrow[r]{i/o} (q', \kappa')$ is a *discrete transition*, if
  - $q \xrightarrow[r]{[g]i/o} q' \in \mathit{runs}(\mathcal{M})$,
  - κ satisfies the conjunction of $\mathit{Inv}(q)$ and g,
  - $\kappa'(x)$ is 0 if x is in r, and $\kappa(x)$ otherwise, for all clocks x, and
  - κ' satisfies $\mathit{Inv}(q')$.

A *timed run* of $\mathcal{M}$ is a sequence of configuration transitions such that delay and discrete transitions alternate, beginning and ending with a delay transition. The set of all timed runs of $\mathcal{M}$ is denoted $\mathit{truns}(\mathcal{M})$.
The *untimed projection* of a timed run ρ, denoted by $\mathit{untime}(\rho)$, is the run obtained by omitting the valuations and delay transitions of ρ. A run π is said to be *feasible* if there exists a timed run ρ such that $\mathit{untime}(\rho) = \pi$.

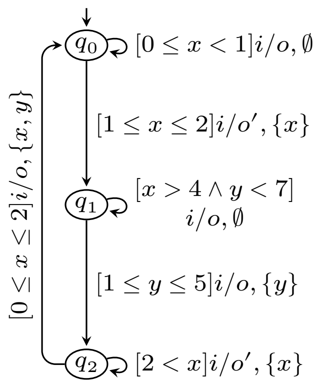
Again, missing symbols in $(q, \kappa) \xrightarrow[u]{i/o} (q', \kappa')$ or $(q, \kappa) \xrightarrow{d} (q, \kappa - d)$ are quantified existentially.

A *timed word* over a set Σ is an alternating sequence of delays from $\mathbb{R}^{\geq 0}$ and symbols from Σ, such that it starts and ends with a delay. The length of a timed word w, denoted by $|w|$, is the number of symbols of Σ in w, *e.g.*, if $|w| = 0$, then $w = d$ with $d \in \mathbb{R}^{\geq 0}$. Note that, when $\Sigma = I$, a timed run reading a timed word w is uniquely determined by its first configuration and w. We thus write $(p, \kappa) \xrightarrow{w}$ for a timed run.

Example 8.2.4. A sample timed run of the sound TMM of Figure 8.1 (repeated in the margin) is

$$\begin{aligned}\rho = (q_0, x = 0, y = 0) &\xrightarrow{0.5} (q_0, x = 0.5, y = 0.5) \xrightarrow[\emptyset]{i/o} (q_0, x = 0.5, y = 0.5)\\ &\xrightarrow{1} (q_0, x = 1.5, y = 1.5) \xrightarrow[\{x\}]{i/o'} (q_1, x = 0, y = 1.5)\\ &\xrightarrow{2} (q_1, x = 2, y = 3.5) \xrightarrow[\{y\}]{i/o} (q_2, x = 2, y = 0)\\ &\xrightarrow{0} (q_2, x = 2, y = 0) \xrightarrow[\{x,y\}]{i/o} (q_0, x = 0, y = 0) \xrightarrow{2} (q_0, x = 2, y = 2).\end{aligned}$$

Then, *untime*(ρ) is the run π of Example 8.2.2, *i.e.*, π is feasible. Notice the transitions with a null delay, indicating that two actions occur "at the same time". Finally, all valuations satisfy the invariants of their corresponding state.
The run $q_0 \xrightarrow{i} q_1 \xrightarrow{i} q_1$ is not feasible. Indeed, it is impossible to reach a valuation satisfying the guard $x > 4 \land y < 7$, as the invariant of q_1 is $0 \leq x \leq 3 \land 0 \leq y \leq 5$. That is, the maximal value of x in q_1 is 3, which does not satisfy $x > 4$.


q_0
$[0 \leq x < 1]i/o, \emptyset$
$[1 \leq x \leq 2]i/o', \{x\}$
q_1
$[x > 4 \land y < 7]$
$i/o, \emptyset$
$[1 \leq y \leq 5]i/o, \{y\}$
q_2
$[2 < x]i/o', \{x\}$
$[0 \leq x \leq 2]i/o, \{x, y\}$

The classical notion of timed automaton (and its semantics) can easily be obtained from the above definitions: drop outputs and add a set of final states F.[4] The definitions of (timed) runs do not need to change (apart from the absence of outputs). Then, a timed run is accepting if the state in the last configuration is final. The language of an automaton is the set of all timed words that label an accepting timed run.

4: To construct a timed automaton from a TMM, simply drop the outputs and set every state to be final.

8.3. Reachability and regions

Let us introduce the *reachability problem* which, given a TMM $\mathcal{M}$ and a state q, asks whether there exists a valuation κ such that $(q_0^{\mathcal{M}}, \kappa_0) \to (q, \kappa)$ is a timed run of $\mathcal{M}$. For timed automata (and, thus, for timed Mealy machines), it is known to be PSPACE-complete [AD94; AL02; WDR13].

[AD94]: Alur et al. (1994), "A Theory of Timed Automata"
[AL02]: Aceto et al. (2002), "Is your model checker on time? On the complexity of model checking for timed modal logics"
[WDR13]: Waez et al. (2013), "A survey of timed automata for the development of real-time systems"

Theorem 8.3.1 ([AD94]). *The reachability problem for TMMs is* PSPACE-*complete.*

In short, the hardness comes from a reduction from the acceptance problem for *Linear-bounded Turing machine* (*LBTM*, for short). We do not give the proof for TMMs here. In Section 9.5, we rely on a similar approach for Mealy machines with timers and provide the details there.

The membership can be obtained by grouping together configurations that share the same timed behavior into *regions*.

Definition 8.3.2 (Clock region [AD94]). Let $\mathcal{M} = (I, O, C, Q, q_0, Inv, \delta)$ be a TMM. For a clock $x \in C$, m_x denotes the largest constant against which x is compared in the invariants or guards of $\mathcal{M}$. Two valuations κ and κ' are *clock-equivalent*, denoted by $\kappa \cong \kappa'$, if

- for all $x \in C$, $\lfloor \kappa(x) \rfloor = \lfloor \kappa'(x) \rfloor$ or $\kappa(x), \kappa'(x) > m_x$, and
- for all $x \in C$ with $\kappa(x) \leq m_x$, $\mathsf{frac}(\kappa(x)) = 0$ if and only if $\mathsf{frac}(\kappa'(x)) = 0$, and
- for all $x_1, x_2 \in C$ with $\kappa(x_1) \leq m_{x_1}$ and $\kappa(x_2) \leq m_{x_2}$, $\mathsf{frac}(\kappa(x_1)) \leq \mathsf{frac}(\kappa(x_2))$ if and only if $\mathsf{frac}(\kappa'(x_1)) \leq \mathsf{frac}(\kappa'(x_2))$.

A *clock region* for $\mathcal{M}$ is an equivalence class of clock valuations induced by $\cong$. We lift the relation to configurations: $(q, \kappa) \cong (q', \kappa')$ if and only if $\kappa \cong \kappa'$ and $q = q'$.

One can show that the region equivalence is a *(strong) time-abstracting bisimulation* [Cer92; TY01; Cla+18]:

[Cer92]: Cerans (1992), "Decidability of Bisimulation Equivalences for Parallel Timer Processes"
[TY01]: Tripakis et al. (2001), "Analysis of Timed Systems Using Time-Abstracting Bisimulations"
[Cla+18]: Clarke et al. (2018), *Handbook of Model Checking*

Definition 8.3.3 (Timed-abstracted bisimulation [Cer92]). A relation R on the states $Q^{\mathcal{M}}$ of $\mathcal{M}$ is a *time-abstracted bisimulation* if, for every $i \in I$, and configurations (q_1, κ_1) and (q_2, κ_2),

- $(q_1, \kappa_1) \; R \; (q_2, \kappa_2)$ and
- $(q_1, \kappa_1) \xrightarrow{d_1 \cdot i} (q_1', \kappa_1')$ for some delay $d_1 \in \mathbb{R}^{\geq 0}$

imply

- $(q_2, \kappa_2) \xrightarrow{d_2 \cdot i} (q_2', \kappa_2')$ for some $d_2 \in \mathbb{R}^{\geq 0}$ and
- $(q_1', \kappa_1') \; R \; (q_2', \kappa_2')$,

and vice-versa.

From the equivalence classes of $\cong$, one can build a finite automaton (or Mealy machine if outputs are kept), called the *region automaton* [AD94].[5] This automaton uses the input symbol of $\mathcal{M}$ and also a special symbol τ that abstracts a delay transition: while a discrete transition allows to change state and, potentially, reset some clocks, reading τ is akin to let time elapse: we go from the state $[\![(q, \kappa)]\!]_{\cong}$ to $[\![(q, \kappa')]\!]_{\cong}$ if $(q, \kappa) \xrightarrow{d} (q, \kappa')$ for some $d \in \mathbb{R}^{\geq 0}$. Finally, the initial state is the equivalence class of $(q_0^{\mathcal{M}}, \kappa_0)$ where $\kappa_0(x) = 0$ for all clocks x. Let us denote this automaton by $\mathcal{R}(\mathcal{M})$.

5: We do not explicitly give the definition for clock-regions. However, in Section 9.5, we give a construction for *timer*-regions.

It is possible to show that a state $[\![(q, \kappa)]\!]_{\cong}$ of $\mathcal{R}(\mathcal{M})$ is reachable if and only if (q, κ) is reachable by a timed run of $\mathcal{M}$ [Cla+18]. While the region automaton has exponentially many states in the number of clocks, it is possible to compute a run on the fly, hence, in polynomial time. We immediately obtain the PSPACE membership of Theorem 8.3.1.

Finally, regions can be used as a tool for other decision problems, such as untimed language[6] equivalence and language equivalence of two timed automata [Cla+18].

6: The untimed projection of the language of a timed automaton.

8.4. Zones

While regions offer an interesting and useful abstraction of the infinite state space of the semantics of a TMM, the number of states in the region automaton is exponential in the number of clocks and the maximal constants appearing in the guards. Hence, efforts have been made to develop more efficient representations of the state space [Hen+92] by using *zones*. In short, zones form a coarser relation over the state space. We here give the definition of zones and some of their properties and refer to [BY03; Bou+22] for a more complete introduction.

[Hen+92]: Henzinger et al. (1992), "Symbolic Model Checking for Real-time Systems"

[BY03]: Bengtsson et al. (2003), "Timed Automata: Semantics, Algorithms and Tools"

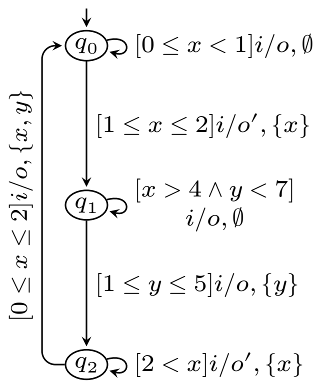
[Bou+22]: Bouyer et al. (2022), "Zone-Based Verification of Timed Automata: Extrapolations, Simulations and What Next?"

Definition 8.4.1 (Zones [Hen+92])**.** Let C be a set of clocks. A subset Z of the valuations over C is called a *zone* if there exists $\phi \in \Phi(C)$ such that Z is exactly the set of valuations that satisfy ϕ.
We define the following operations on zones:

- The *upward closure* of Z where we let some time elapsed in all valuations of Z. That is, we obtain all valuations that can be reached from Z by just waiting, *i. e.*,
$$Z\!\uparrow = \{\kappa + d \mid \kappa \in Z, d \in \mathbb{R}^{\geq 0}\}.$$
- The *reset* of Z under $r \subseteq C$ is obtained by resetting the clocks r in each valuation of Z, *i. e.*,
$$Z[r] = \{\kappa \mid \exists \kappa' \in Z, \forall x \in r, y \in C \setminus r : \kappa(x) = 0 \wedge \kappa(y) = \kappa'(y).\}$$

One can show that $Z\!\uparrow$, $Z[r]$, and $Z \cap Z'$ are zones over C, when Z and Z' are zones and $r \subseteq C$ [Hen+92]. Zones can be efficiently computed via *difference bound matrices* (DBM) [Cla+18; BY03; Bou+22]. Moreover, a *zone timed Mealy machine* (or zone timed automaton) can be constructed, which we denote $\mathit{zone}(\mathcal{M})$. Instead of providing a formal definition for $\mathit{zone}(\mathcal{M})$, we give an example.

Example 8.4.2. Let $\mathcal{M}$ be the TMM of Figure 8.1, which is repeated in the margin. Recall that $\mathit{Inv}(q_0) = 0 \leq x \leq 2$, $\mathit{Inv}(q_1) = 0 \leq x \leq 3 \wedge 0 \leq y \leq 5$, and $\mathit{Inv}(q_2) = 0 \leq y \leq 1$. We construct the zone TMM $\mathit{zone}(\mathcal{M})$. Rather, we construct the zone Mealy machine (the guards and invariants can be copied from $\mathcal{M}$ to obtain a TMM).
The initial state is obtained by letting time elapse in the initial configuration (q_0, κ_0) (such that $\kappa_0(c) = 0$ for all clocks c) and restricting the set of valuations to those that satisfy the invariant of q_0. That is, we first compute the upward closure of $\{\kappa_0\}$, which is the set $\{\kappa_0\}\!\uparrow = \{\kappa_0 + d \mid d \in \mathbb{R}^{\geq 0}\}$. We then only keep the valuations that satisfy $0 \leq x \leq 2$, *i. e.*, every valuation κ such that $0 \leq \kappa(x) = \kappa(y) \leq 2$. For simplicity, we denote this set by $0 \leq x = y \leq 2$. The initial state of $\mathit{zone}(\mathcal{M})$ is then $(q_0, 0 \leq x = y \leq 2)$.
Let us define its outgoing transitions. We start with the transition $q_0 \xrightarrow[\emptyset]{[0 \leq x < 1] i/o} q_0$ of $\mathcal{M}$. First, we compute the intersection of $0 \leq x = y \leq 2$ and $0 \leq x < 1$ (to keep the valuations that can trigger the transition), which yields the zone $0 \leq x = y < 1$. Then, we again let time elapse and keep the valuations that satisfy $\mathit{Inv}(q_0)$. That is, we define the transition
$$(q_0, 0 \leq x = y \leq 2) \xrightarrow{i/o} (q_0, 0 \leq x = y \leq 2).^{7}$$


q_0
$[0 \leq x < 1] i/o, \emptyset$
$[1 \leq x \leq 2] i/o', \{x\}$
q_1
$[x > 4 \wedge y < 7]$
$i/o, \emptyset$
$[1 \leq y \leq 5] i/o, \{y\}$
q_2
$[2 < x] i/o', \{x\}$
$[0 \leq x \leq 2] i/o, \{x, y\}$

7: The zone TMMs copied the guard $[0 \leq x < 1]$ and the reset $\emptyset$ from the transition of $\mathcal{M}$.

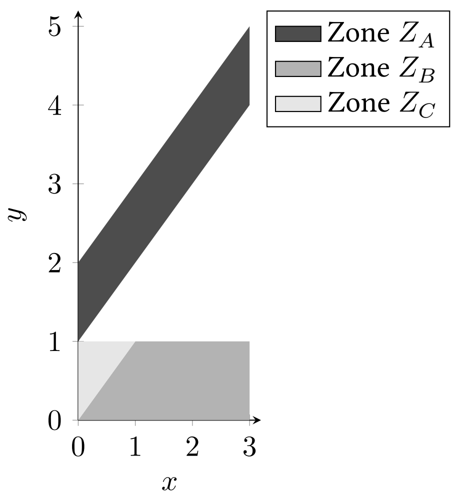


Figure 8.2: Visualization of some of the zones of Example 8.4.2.

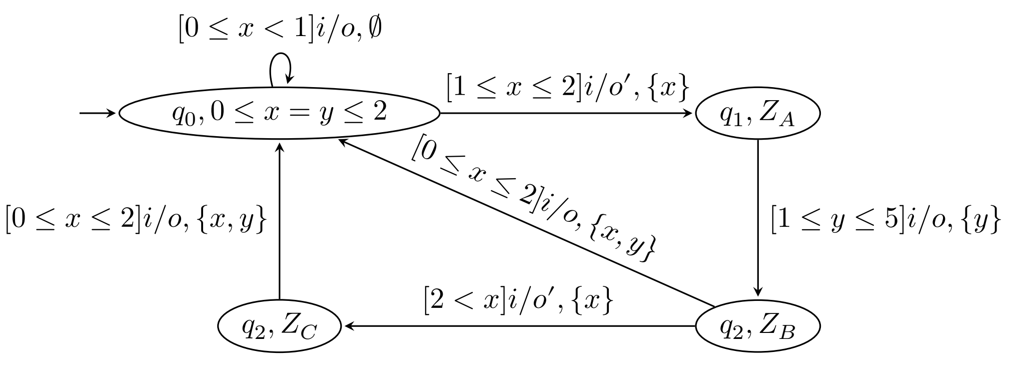

$[0 \leq x < 1]i/o, \emptyset$
$q_0, 0 \leq x = y \leq 2$
$[1 \leq x \leq 2]i/o', \{x\}$
q_1, Z_A
$[0 \leq x \leq 2]i/o, \{x, y\}$
$[0 \leq x \leq 2]i/o, \{x, y\}$
$[1 \leq y \leq 5]i/o, \{y\}$
$[2 < x]i/o', \{x\}$
q_2, Z_C
q_2, Z_B

Figure 8.3: The zone TMM of the TMM of Figure 8.1. The zones Z_A, Z_B, and Z_C are depicted in Figure 8.2.

Let us now consider the transition $q_0 \xrightarrow[\{x\}]{[1 \leq x \leq 2]i/o'} q_1$ of $\mathcal{M}$. Again, we restrict the valuations of the zone $0 \leq x = y \leq 2$ to those that satisfy $1 \leq x \leq 2$, *i.e.*, the set given by $1 \leq x = y \leq 2$. The transition of $\mathcal{M}$ restarts x. So, after taking the transition, we are in the zone $x = 0 \land 1 \leq y \leq 2$. It remains to let time elapse and take the intersection with $\mathit{Inv}(q_1)$. We obtain the zone

$$Z_A = (0 \leq x \leq 3) \land (1 \leq y \leq 5) \land (-2 \leq x - y \leq -1).$$

This zone is represented by the darkest area of Figure 8.2. That is, we define the transition

$$(q_0, 0 \leq x = y \leq 2) \xrightarrow{i/o'} (q_1, Z_A).$$

We can continue this process and define states and transitions as we discover them. Figure 8.3 gives the resulting zone TMM.
Observe that the self-loop over q_1 is not reproduced in $\mathit{zone}(\mathcal{M})$. This is due to the fact that the intersection of Z_A and $(x > 4 \land y < 7)$ is empty. That is, we only define transitions when the target zone is non-empty. In other words, computing the zone TMM allows to "prune" the transitions that are never feasible. Even better, one can show that every run of $\mathit{zone}(\mathcal{M})$ is feasible.

8.5. Other results

To conclude this introduction to timed machines, let us list some known properties. (We refer to [WDR13] for more properties.) We call a language L *timed regular* if there exists a timed automaton[9] $\mathcal{A}$ that accepts L [AD94]. The first theorem gives properties over the set of timed regular languages.

[WDR13]: Waez et al. (2013), "A survey of timed automata for the development of real-time systems"

9: Recall that an automaton can be obtained from a Mealy machine by dropping the outputs and setting every state to be final.

[AD94]: Alur et al. (1994), "A Theory of Timed Automata"

Theorem 8.5.1 ([AD94]). *The class of timed regular languages is closed under union, intersection, concatenation, and Kleene-star.*
The untimed language of a timed regular language is a regular language.

One could need to check whether a timed regular language is empty or not, which is a decidable problem.

Theorem 8.5.2 ([AD94]). *The emptiness problem for timed automata is* PSPACE*-complete.*

There are also problems that are known to be undecidable, when considering timed automata in general (*i. e.*, without any restrictions).

Theorem 8.5.3 ([AD94]). *Given two timed automata $\mathcal{A}$ and $\mathcal{A}'$, all of the following decision problems are undecidable:*

- *Are the languages of $\mathcal{A}$ and $\mathcal{A}'$ equal?*
- *Is the language of $\mathcal{A}$ included in the language of $\mathcal{A}'$?*
- *Does $\mathcal{A}$ accept any timed word?*[10]

10: This is known as the universality problem.

From the undecidability of the inclusion problem, we immediately obtain that the family of timed regular languages is not closed under complement [AD94].

While we refrain from defining this subfamily here, it is noteworthy that some of the problems become decidable for *deterministic* timed automata. For instance, the inclusion and equivalence problems for deterministic automata become PSPACE-complete [AD94].

Mealy Machines with Timers 9.

In this chapter, based on [Bru+23; Bru+24], we introduce a subfamily of timed Mealy machines, called *Mealy machines with timers*, in which timing constraints are given by setting a timer to a given value. Later on, this timer will eventually reach zero, provoking a special *timeout* symbol. It may happen that a timer times out at the same time an input is provided, or that two timers reach zero at the same time, leading to a non-deterministic behavior of the machine, as it can arbitrarily decide which action to process first.

[Bru+23]: Bruyère et al. (2023), "Automata with Timers"
[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

We study many problems in this chapter: how to decide whether the Mealy machines are equivalent, what is the complexity of deciding whether a state is reachable by some timed run, and whether every untimed behavior of a machine can be witnessed via timed runs in which all delays are positive. Furthermore, we highlight that a timed Mealy machine (see the previous chapter) can be constructed from a Mealy machine with timers, *i.e.*, the model we present here is indeed a subfamily of timed Mealy machines.

Technical proofs and details are deferred to Appendix C.

Chapter contents

9.1. Introduction

As said in the previous chapter, timed automata were introduced by Alur & Dill [AD94] as finite-state automata equipped with real-valued clock variables

[AD94]: Alur et al. (1994), "A Theory of Timed Automata" for measuring the time between state transitions. These clock variables all increase at the same rate when time elapses, they can be reset along transitions, and be used in guards along transitions and in invariant predicates for states. Timed automata have become a framework of choice for modeling and analysis of real-time systems, equipped with a rich theory, supported by powerful tools, and with numerous applications [Bou+18]. In the previous chapter, we extended timed automata to timed Mealy machines, *i.e.*, timed automata with outputs.

[Bou+18]: Bouyer et al. (2018), "Model Checking Real-Time Systems"

Interestingly, whereas the values of clocks in a timed automaton *increase* over time, designers of real-time systems (e.g. embedded controllers and network protocols) typically use timers to enforce timing constraints, and the values of these timers *decrease* over time. If an application starts a timer with a certain value t, then this value decreases over time and after t time units — when the value has become 0 — a timeout event occurs. It is straightforward to encode the behavior of timers using a timed automaton. Timed automata allow one to express a richer class of behaviors than what can be described using timers, and can for instance express that the time between two events is contained in an interval $[t-d, t+d]$. Moreover, timed automata can express constraints on the timing between arbitrary events, not just between start and timeout of timers.

However, the expressive power of timed automata entails certain problems. For instance, one can easily define timed automata models in which time stops at some point (timelocks) or an infinite number of discrete transitions occurs in a finite time (Zeno behavior). Thus timed automata may describe behavior that cannot be realized by any physical system. Also, learning [Ang87; HS18] of timed automata models in a black-box setting turns out to be challenging [GJP06; GJL10; An+20; Wag23]. For a learner who can only observe the external events of a system and their timing, it may be really difficult to infer the logical predicates (invariants and guards) that label the states and transitions of a timed automaton model of this system. As a result, all known learning algorithms for timed automata suffer from combinatorial explosions, which severely limits their practical usefulness.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"
[HS18]: Howar et al. (2018), "Active Automata Learning in Practice - An Annotated Bibliography of the Years 2011 to 2016"
[GJP06]: Grinchtein et al. (2006), "Inference of Event-Recording Automata Using Timed Decision Trees"
[GJL10]: Grinchtein et al. (2010), "Learning of event-recording automata"
[An+20]: An et al. (2020), "Learning One-Clock Timed Automata"
[Wag23]: Waga (2023), "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization"

For these reasons, it is interesting to consider variations of timed automata (rather, of timed Mealy machines here) whose expressivity is restricted by using timers instead of clocks, as introduced by Dill in [Dil89]. In a Mealy machine with timers (MMT), a transition may start a timer by setting it to a certain value. Whenever a timer reaches zero, it produces an observable timeout symbol that triggers a transition in the automaton. Dill also shows that the space of timer valuations can be abstracted into a finite number of so-called *regions*. However, the model we study here is slightly different, as, unlike Dill, we allow a timer to be stopped before reaching zero. We also study the regions of our MMT and give an upper bound on their number.

[Dil89]: Dill (1989), "Timing Assumptions and Verification of Finite-State Concurrent Systems"

Vaandrager *et al.* [VBE21] provide a black-box active learning algorithm for Mealy machines with a single timer, and evaluate it on a number of realistic applications, showing that it outperforms the timed automata based approaches of Aichernig *et al.* [APT20] and An *et al.* [An+20]. Even though many realistic systems can be modeled as MM1Ts (*e.g.*, the benchmarks described in [VBE21] and the brick sorter and traffic controller examples in [Die+23]), the restriction

[VBE21]: Vaandrager et al. (2021), "Learning Mealy Machines with One Timer"
[APT20]: Aichernig et al. (2020), "From Passive to Active: Learning Timed Automata Efficiently"
[An+20]: An et al. (2020), "Learning One-Clock Timed Automata"
[Die+23]: Dierl et al. (2023), "Learning Symbolic Timed Models from Concrete Timed Data"

to a single timer is a serious limitation. Therefore, Kogel *et al.* [KKG23] propose *Mealy machines with local timers (MMLTs)*, an extension of Mealy machines with one timer to multiple timers subject to carefully chosen constraints to enable efficient learning. Although quite interesting, the constraints of MMLTs are too restrictive for many applications (*e.g.*, the FDDI protocol presented in the next chapter). Also, any MMLT can be converted to an equivalent MM1T. If we want to extend the learning algorithm of [VBE21] to a setting with multiple (non-local) timers, we need to deal with the issue of *races*, *i.e.*, situations where multiple timers reach zero (and thus timeout) simultaneously. If a race occurs, then (despite the automaton being deterministic!) the automaton can process the simultaneous timeouts in various orders, leading to nondeterministic behavior.

[KKG23]: Kogel et al. (2023), "Learning Mealy Machines with Local Timers"

An active learning algorithm for MMTs is presented in the next chapter. We here formally define the model and study multiple problems:

- We show that a timed Mealy machine (TMM) can be constructed from an MMT in Section 9.3.
- In Section 9.4, we define two ways of testing equivalence between two MMTs: one that focuses on timed runs, and one that remains in the untimed world. We show that the latter implies the first, *i.e.*, it is sufficient to test the untimed equivalence.
- In Section 9.5, we adapt to MMTs the reachability problem and the notion of *regions* that we introduced for TMMs in Section 8.3.
- As said above, races lead to nondeterministic behaviors. We thus provide in Section 9.6 an algorithm that can decide whether every untimed behavior of an MMT can be observed via timed runs without races, *i.e.*, whether races are required.
- Finally, in Section 9.7, we adapt to MMTs the notion of *zones* introduced in Section 8.4 for TMMs.

Technical details and proofs are deferred to Appendix C.

9.2. Mealy machines with timers

A Mealy machine with timers is a Mealy machine augmented with a finite number of timers, which can be used to enforce timing constraints over the behavior of the machine, *e.g.*, if we sent a message and we did not receive the acknowledgment after d units of time, we resend the message. In this section, we first properly define Mealy machines with timers, alongside their (timed) semantics. We then introduce in Section 9.2.2 the notion of blocks, which abstract timed runs. Finally, Section 9.2.3 discusses *enabled* timers (*i.e.*, timers that can time out after reading a run), which allows us to define when an MMT is complete.

Given a *timer* x, we write $\mathit{to}[x]$ for its timeout symbol. Then, if X is a set of timers, $\mathit{TO}[X]$ is the set of all timeout symbols, *i.e.*,

$$\mathit{TO}[X] = \{\mathit{to}[x] \mid x \in X\}.$$

Furthermore, if $\mathcal{M}$ is a Mealy machine using X as its set of timers and I as its set of input symbols, we denote by $A(\mathcal{M})$ the set of *actions of* $\mathcal{M}$: reading an input (an *input action*), or processing a timeout (a *timeout action*), *i.e.*,

$$A(\mathcal{M}) = I \cup TO[X].$$

Finally, the set of *updates of* $\mathcal{M}$ is

$$U(\mathcal{M}) = (X \times \mathbb{N}^{>0}) \cup \{\bot\},$$

where (x, c) means that the timer x is started with value c, and $\bot$ stands for no timer update.

We impose certain constraints on the shape of our machines:

- a timer must be explicitly started to become active,
- a timer x can time out only when it is active, and
- a *to*$[x]$-transition may only restart x (if the update is not $\bot$).

These restrictions allow us to define hereafter the timed semantics in a straightforward approach.

Definition 9.2.1 (Mealy machine with timers)**.** A *Mealy machine with timers* (*MMT*, for short) is a tuple $\mathcal{M} = (I, O, X, Q, q_0, \chi, \delta)$ where:

- I is an input alphabet, O an output alphabet, and X a finite set of timers,
- Q is a finite set of states, with $q_0 \in Q$ the initial state,
- $\chi : Q \to \mathcal{P}(X)$ is a (total) function that assigns a set of *active* timers to each state, and
- $\delta : Q \times A(\mathcal{M}) \rightharpoonup Q \times O \times U(\mathcal{M})$ is a (partial) transition function that assigns a state-output-update triple to a state-action pair. We write $q \xrightarrow[u]{i/o} q'$ if $\delta(q, i) = (q', o, u)$.

An MMT $\mathcal{M}$ is *sound* if for all $q, q' \in Q$, $i \in A(\mathcal{M})$, $o \in O$, $x \in X$, $c \in \mathbb{N}^{>0}$:

$$\begin{aligned}\chi(q_0) &= \emptyset\\ q \xrightarrow[\bot]{i/o} q' &\Rightarrow \chi(q') \subseteq \chi(q)\\ q \xrightarrow[(x,c)]{i/o} q' &\Rightarrow x \in \chi(q') \land \chi(q') \setminus \{x\} \subseteq \chi(q)\\ q \xrightarrow[\bot]{to[x]/o} q' &\Rightarrow x \in \chi(q) \land x \notin \chi(q')\\ q \xrightarrow[(y,c)]{to[x]/o} q' &\Rightarrow x \in \chi(q) \land x = y.\end{aligned}$$

As usual, we add, when needed, a superscript to indicate which MMT is considered, *e.g.*, $Q^{\mathcal{M}}, q_0^{\mathcal{M}}$, etc. Missing symbols in $q \xrightarrow[u]{i/o} q'$ are quantified existentially, *e.g.*, $q \xrightarrow[u]{i/o}$ means there exists q' such that $q \xrightarrow[u]{i/o} q'$, and $q \xrightarrow{i}$ means there exist o and u such that $q \xrightarrow[u]{i/o}$. We say that a transition $q \xrightarrow[u]{i} q'$

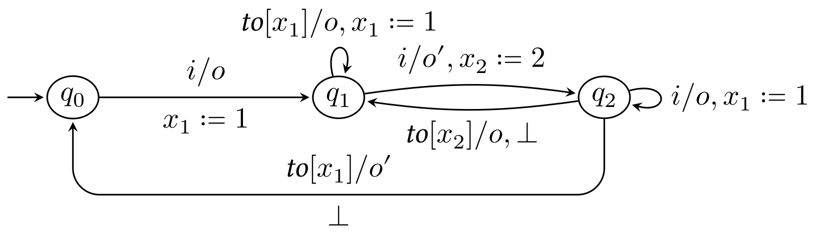


Figure 9.1: An MMT with $\chi(q_0) = \emptyset$, $\chi(q_1) = \{x_1\}$, $\chi(q_2) = \{x_1, x_2\}$.

- *starts* (resp. *restarts*) the timer x if $u = (x, c)$ and x is inactive (resp. active) in q,
- *stops* the timer x if $i \neq to[x]$ and x is inactive in q',[1] and
- *discards* x if it stops or restarts x.

1: Observe that a transition cannot (re)start a timer and stop the same timer.

A *run* π of $\mathcal{M}$ either consists of a single state p_0 or of a nonempty sequence of transitions

$$\pi = p_0 \xrightarrow[u_1]{i_1/o_1} p_1 \xrightarrow[u_2]{i_2/o_2} \cdots \xrightarrow[u_n]{i_n/o_n} p_n.$$

We denote by *runs*$(\mathcal{M})$ the set of runs of $\mathcal{M}$. We often write $q \xrightarrow{i} \in runs(\mathcal{M})$ to highlight that $\delta(q, i)$ is defined. We lift the notation to words $i_1 \cdots i_n$ as usual: $p_0 \xrightarrow{i_1 \cdots i_n} p_n \in runs(\mathcal{M})$ if there exists a run $p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_n} p_n \in runs(\mathcal{M})$. Note that any run π is uniquely determined by its first state p_0 and word, as $\mathcal{M}$ is deterministic.

Definition 9.2.2 (x-spanning run). A run $p_0 \xrightarrow[u_1]{i_1} \cdots \xrightarrow[u_n]{i_n} p_n$ is termed x-*spanning* (with $x \in X$) if it begins with a transition (re)starting x, ends with a $to[x]$-transition, and no intermediate transition restarts or stops x. That is,

- $u_1 = (x, c)$,
- $i_n = to[x]$,
- $u_j \neq (x, d)$ for all $j \in \{2, \dots, n-1\}$ and $d \in \mathbb{N}^{>0}$, and
- $x \in \chi(p_j)$ for all $j \in \{2, \dots, n-1\}$.

Example 9.2.3. Figure 9.1 shows an MMT $\mathcal{M}$ with timers $X = \{x_1, x_2\}$, inputs $I = \{i\}$, and outputs $O = \{o, o'\}$. In the initial state q_0, no timer is active, while x_1 is active in q_1 and q_2, and x_2 is active in q_2. Timer updates are shown on the transitions. For instance, x_1 is started with value 1 when going from q_0 to q_1. The transition $q_2 \xrightarrow{i} q_2$ restarts x_1 with value 1. A sample run is

$$\gamma = q_0 \xrightarrow[(x_1,1)]{i/o} q_1 \xrightarrow[(x_2,2)]{i/o'} q_2 \xrightarrow[\perp]{to[x_1]/o'} q_0 \in runs(\mathcal{M}).$$

which can be written as $\gamma = q_0 \xrightarrow{i \cdot i \cdot to[x_1]} q_0$. The run

$$q_0 \xrightarrow[(x_1,1)]{i} q_1 \xrightarrow[(x_2,2)]{i} q_2 \xrightarrow{to[x_1]} q_0$$

is x_1-spanning, while

$$q_2 \xrightarrow[(x_1,1)]{i} q_2 \xrightarrow[(x_1,1)]{i} q_2 \xrightarrow{to[x_1]} q_0$$

is not as the second transition restarts x_1.

9.2.1. Timed semantics

Let us now define the timed semantics of an MMT, via an infinite-state labeled transition system describing all possible configurations and transitions between them.

A *valuation* is a partial function $\kappa\colon X \rightharpoonup \mathbb{R}^{\geq 0}$ that assigns nonnegative real numbers to timers. For $Y \subseteq X$, we write $\mathsf{Val}(Y)$ for the set of all valuations κ with $\mathrm{dom}(\kappa) = Y$. A *configuration* of $\mathcal{M}$ is a pair (q, κ) where $q \in Q$ and $\kappa \in \mathsf{Val}(\chi(q))$. The *initial configuration* is the pair (q_0, κ_0) where $\kappa_0 = \emptyset$ since $\chi(q_0) = \emptyset$. If $\kappa \in \mathsf{Val}(Y)$ is a valuation in which all timers from Y have a value of at least $d \in \mathbb{R}^{\geq 0}$, then d units of time may *elapse*. We write $\kappa - d \in \mathsf{Val}(Y)$ for the resulting valuation that satisfies $(\kappa - d)(x) = \kappa(x) - d$, for all $x \in Y$. If the valuation κ contains a value $\kappa(x) = 0$ for some timer x, then x may *time out*.

Definition 9.2.4 (Timed run)**.** We define the transitions between configurations $(q, \kappa), (q', \kappa')$ as follows.

- ▶ $(q, \kappa) \xrightarrow{d} (q, \kappa')$ is a *delay transition* if
  - $\kappa(x) \geq d$ for every $x \in \chi(q)$, and
  - $\kappa' = \kappa - d$.
- ▶ $(q, \kappa) \xrightarrow[u]{i/o} (q', \kappa')$ is a *discrete transition*, if
  - $q \xrightarrow[u]{i/o} q' \in \mathit{runs}(\mathcal{M})$,
  - if $u = (x, c)$, then
    - ⋆ $\kappa'(x) = c$, and
    - ⋆ for every $y \in \chi(q')$ such that $u \neq (y, \cdot)$, $\kappa'(y) = \kappa(y)$, and
  - $\kappa(x) = 0$ if $i = \mathit{to}[x]$.

  Moreover, if $i = \mathit{to}[x]$, the transition is a *timeout transition*. Otherwise, it is an *input transition*.

A *timed run* of $\mathcal{M}$ is a sequence of configuration transitions such that delay and discrete transitions alternate, beginning and ending with a delay transition. The set of all timed runs of $\mathcal{M}$ is denoted $\mathit{truns}(\mathcal{M})$.
The *untimed projection* of a timed run ρ, denoted by $\mathit{untime}(\rho)$, is the run obtained by omitting the valuations and delay transitions of ρ. A run π is termed *feasible* if there exists a timed run ρ such that $\mathit{untime}(\rho) = \pi$. Finally, $\mathcal{M}$ is *feasible* if every run of $\mathcal{M}$ is feasible.

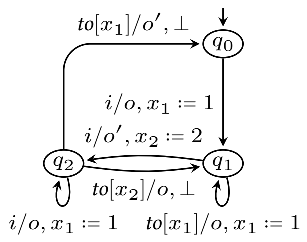
Again, missing symbols in $(q, \kappa) \xrightarrow[u]{i/o} (q', \kappa')$ or $(q, \kappa) \xrightarrow{d} (q, \kappa - d)$ are quantified existentially.

A *timed word* over a set Σ is an alternating sequence of delays from $\mathbb{R}^{\geq 0}$ and symbols from Σ, such that it starts and ends with a delay. The length of a timed word w, denoted by $|w|$, is the number of symbols of Σ in w, *e.g.*, if $|w| = 0$, then $w = d$ with $d \in \mathbb{R}^{\geq 0}$. Note that, when $\Sigma = A(\mathcal{M})$, a timed run reading a timed word w is uniquely determined by its first configuration and w. We thus write $(p, \kappa) \xrightarrow{w}$ for a timed run. A timed run ρ is called x*-spanning* (with $x \in X$) if $\mathit{untime}(\rho)$ is x-spanning.

Finally, a *timed output word* (*tow*, for short) is a timed word over O. Given a timed run ρ, we write $\mathit{tow}(\rho)$ for the sequence of alternating delays and output symbols seen along the transitions of ρ. We write $\mathit{toutputs}(w)$ for the set of all tows produced by the timed runs reading the timed word w.

Example 9.2.5. A sample timed run of the sound MMT of Figure 9.1 (repeated in the margin) is

$$\begin{aligned}\rho = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 0, x_2 = 2) \xrightarrow{0} (q_2, x_1 = 0, x_2 = 2)\\ &\xrightarrow[\perp]{\mathit{to}[x_1]/o'} (q_0, \emptyset) \xrightarrow{1.5} (q_0, \emptyset).\end{aligned}$$



$\mathit{to}[x_1]/o', \perp$
q_0
$i/o, x_1 := 1$
$i/o', x_2 := 2$
q_2
q_1
$\mathit{to}[x_2]/o, \perp$
$i/o, x_1 := 1$
$\mathit{to}[x_1]/o, x_1 := 1$

Then, $\mathit{untime}(\rho)$ is the run γ of Example 9.2.3, which implies that γ is feasible. Notice the transitions with a null delay, indicating that two actions occur "at the same time". In such a situation, the MMT can arbitrarily decide the order in which the actions are processed. Here, we first triggered the i-transition before reading $\mathit{to}[x_1]$. An other timed run of the automaton may have done the opposite. Hence, the MMT may non-deterministic behavior. This non-determinism is studied in Section 9.6.

The run $q_0 \xrightarrow{i \cdot i \cdot \mathit{to}[x_2]}$ is not feasible, as any timed run ρ' such that $\mathit{untime}(\rho') = q_0 \xrightarrow{i \cdot i} q_2$ is such that the value of x_2 is strictly greater than the value of x_1. By consequence, x_1 must necessarily time out before x_2 and there is no timed run whose untimed projection is $q_0 \xrightarrow{i \cdot i \cdot \mathit{to}[x_2]}$.

The labels of the delay and discrete transitions of ρ form a timed word over $A(\mathcal{M})$ equal to

$$w = 1 \cdot i \cdot 1 \cdot i \cdot 0 \cdot \mathit{to}[x_1] \cdot 1.5.$$

Since it has four actions, $|w| = 4$. Furthermore,

$$\mathit{tow}(\rho) = 1 \cdot o \cdot 1 \cdot o' \cdot 0 \cdot o' \cdot 1.5.$$

Finally, the timed run $(q_0, \emptyset) \xrightarrow{0.5 \cdot i \cdot 1 \cdot i \cdot 0 \cdot \mathit{to}[x_1] \cdot 2}$ is x_1-spanning, as $q_0 \xrightarrow{i \cdot i \cdot \mathit{to}[x_1]}$ is x_1-spanning.

9.2.2. Blocks

In this section, we abstract the timed runs of an MMT $\mathcal{M}$.[2] To better understand how $\mathcal{M}$ behaves, we decompose timed run ρ (reading the timed word $d_0 \cdot i_0 \cdots i_n \cdot d_{n+1}$) into *blocks*. Recall that a $\mathit{to}[x]$-transition can only occur if there is an

2: For simplicity, we ignore outputs here.

earlier transition (re)starting x. That is, for any $i_k = \mathit{to}[x]$, there must exist $j < k$ such that the update of the transition reading i_j (re)starts x. Blocks then describe which transition (re)starts a timer for which we observe a timeout and the *fate* of the timer (*e.g.*, whether the timer is killed before we can observe its timeout).

Definition 9.2.6 (Block)**.** Let

$$\begin{aligned}\rho = (p_0, \kappa_0) \xrightarrow{d_1} (p_0, \kappa_0 - d_1) \xrightarrow[u_1]{i_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots \\ \xrightarrow[u_n]{i_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1}) \in \mathit{truns}(\mathcal{M})\end{aligned}$$

be a timed run. A *block* of ρ is a pair $B = (k_1 k_2 \dots k_m, \gamma)$ such that $i_{k_1}, i_{k_2}, \dots, i_{k_m}$ is a maximal subsequence of actions of ρ such that

- $i_{k_1} \in I$,
- $p_{k_\ell - 1} \xrightarrow{i_{k_\ell} \cdots i_{k_{\ell+1}}} p_{k_{\ell+1}}$ is x-spanning for some timer x and for all $1 \leq \ell < m$, and
- γ is the *timer fate* of B defined as:

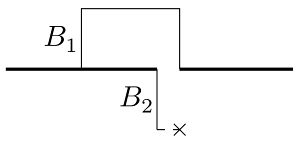
$$\gamma = \begin{cases} \perp & \text{if } i_{k_m} \text{ does not restart any timer} \\ \bullet & \text{if } i_{k_m} \text{ restarts a timer which is discarded (by some } i_\ell, \\ & \text{with } k_m < \ell \leq n \text{ or by the end of the run), when its} \\ & \text{value is zero} \\ \times & \text{otherwise.} \end{cases}$$

In the timer fate definition, consider the case where i_{k_m} restarts a timer x. For the purposes of Section 9.6, it is convenient to know whether x is later discarded or not, and in case it is discarded, whether this occurs when its value is zero (*i.e.*, whether γ is $\times$ or $\bullet$). Furthermore, the information that x is still active in the last configuration (q, κ) of ρ and whether it times out is important. Hence, both $\times$ and $\bullet$ cover the cases occurring when a timed run ends.

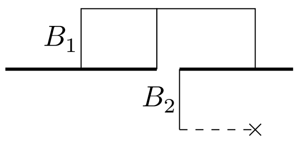
When no confusion is possible, we denote a block by a sequence of inputs rather than the corresponding sequence of indices, *i.e.*, $B = (i_{k_1} i_{k_2} \dots i_{k_m}, \gamma)$. In the sequel, we use notation $i \in B$ to denote an action i belonging to the sequence of B.

By definition of an MMT, recall that the same timer x is restarted along a block B. Hence we also say that B is an x*-block*. Note also that the sequence of a block can be composed of a single input $i \in I$.

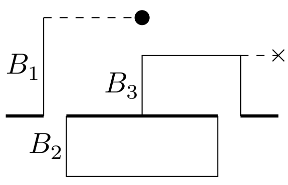
Example 9.2.7. Consider the timed run ρ of Example 9.2.5 from the MMT $\mathcal{M}$



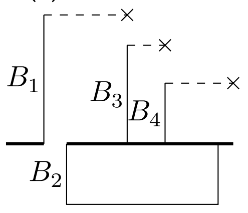
(a) Timed run ρ.



(b) Timed run σ.



(c) Timed run π.

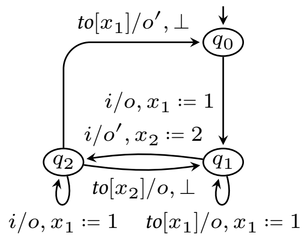


(d) Timed run τ.

Figure 9.2: Block representations of four timed runs.

depicted in Figure 9.1 (and repeated in the margin):

$$\begin{aligned}\rho = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 0, x_2 = 2) \xrightarrow{0} (q_2, x_1 = 0, x_2 = 2)\\ &\xrightarrow[\perp]{to[x_1]/o'} (q_0, \emptyset) \xrightarrow{1.5} (q_0, \emptyset).\end{aligned}$$



It has two blocks: an x_1-block $B_1 = (i\ to[x_1], \perp)$ and an x_2-block $B_2 = (i, \times)$, both represented in Figure 9.2a.[3] In this visual representation of the blocks, time flows left to right and is represented by the thick horizontal line. A "gap" in that line indicates that the time is stopped, *i. e.*, the delay between two consecutive actions is zero. We draw a vertical line for each action, and join together actions belonging to a block by a horizontal (non-thick) line. Moreover, a dotted line finished by $\times$ represents a block whose timer fate is $\times$.

3: When using the action indices in the blocks, we have $B_1 = (1\ 3, \perp)$ and $B_2 = (2, \times)$.

Consider another timed run σ from $\mathcal{A}$:

$$\begin{aligned}\sigma = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\ &\xrightarrow[(x_1,1)]{to[x_1]/o} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 1, x_2 = 2) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1)\\ &\xrightarrow[\perp]{to[x_1]/o'} (q_0, \emptyset) \xrightarrow{0.5} (q_0, \emptyset).\end{aligned}$$

This timed run has also two blocks represented in Figure 9.2b, such that $B_1 = (i\ to[x_1]\ to[x_1], \perp)$ with x_1 timing out twice.
We conclude this example with two other timed runs, π and τ, such that

some of their blocks have a timer fate $\gamma \neq \perp$. Let π and τ be the timed runs:

$$\begin{aligned}\pi = (q_0, \emptyset) &\xrightarrow{0.5} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 1, x_2 = 2) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1)\\ &\xrightarrow[(x_1,1)]{i/o} (q_2, x_1 = 1, x_2 = 1) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 0)\\ &\xrightarrow[\perp]{to[x_2]/o} (q_1, x_1 = 0) \xrightarrow{0} (q_1, x_1 = 0)\\ &\xrightarrow[(x_1,1)]{to[x_1]/o} (q_1, x_1 = 1) \xrightarrow{0.5} (q_1, x_1 = 0.5)\end{aligned}$$

and

$$\begin{aligned}\tau = (q_0, \emptyset) &\xrightarrow{0.5} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 1, x_2 = 2)\\ &\xrightarrow{0.8} (q_2, x_1 = 0.2, x_2 = 1.2)\\ &\xrightarrow[(x_1,1)]{i/o} (q_2, x_1 = 1, x_2 = 1.2)\\ &\xrightarrow{0.5} (q_2, x_1 = 0.5, x_2 = 0.7)\\ &\xrightarrow[(x_1,1)]{i/o} (q_2, x_1 = 1, x_2 = 0.7)\\ &\xrightarrow{0.7} (q_2, x_1 = 0.3, x_2 = 0)\\ &\xrightarrow[\perp]{to[x_2]/o} (q_1, x_1 = 0.3) \xrightarrow{0.2} (q_1, x_1 = 0.1).\end{aligned}$$

The run π has three blocks $B_1 = (i, \bullet)$ (x_1 is started by i and then discarded while its value is zero), $B_2 = (i\ to[x_2], \perp)$, and $B_3 = (i\ to[x_1], \times)$ (x_1 is again started in B_3 but π ends before x_1 reaches value zero). Those blocks are represented in Figure 9.2c, where we visually represent the timer fate of B_1 by a dotted line finished by $\bullet$. Finally, the run τ has its blocks depicted in Figure 9.2d. This time, x_1 is discarded before reaching zero, *i.e.*, $B_1 = (i, \times)$.

As illustrated by the previous example, blocks satisfy the following property.[5]

5: Recall that the sequence of a block can be composed of a single action.

Lemma 9.2.8. *Let*

$$\begin{aligned}\rho = (p_0, \kappa_0) &\xrightarrow{d_1} (p_0, \kappa_0 - d_1) \xrightarrow[u_1]{i_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots\\ &\xrightarrow[u_n]{i_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1}) \in \mathit{truns}(\mathcal{M})\end{aligned}$$

be a timed run an MMT. Then, the sequences of the blocks of ρ form a partition of the set of indices $\{1, \dots, n\}$ of the actions of ρ.

9.2.3. Enabled timers and complete machines

Let us now move towards defining when a machine is deemed complete. Intuitively, we require that, if a timer x can time out in a state q (via some timed run), then $q \xrightarrow{to[x]}$ must be defined. In order to properly define this, we first introduce *enabled timers*, which are those timers that can time out.

Definition 9.2.9 (Enabled timers). Given a timed run

$$\begin{aligned}\rho = (q_0^{\mathcal{M}}, \emptyset) \xrightarrow{d_1} (q_0^{\mathcal{M}}, \emptyset - d_1) \xrightarrow{i_1} (q_1, \kappa_1) \xrightarrow{d_2} \cdots \\ \xrightarrow{i_n} (q_n, \kappa_n) \xrightarrow{d_{n+1}} (q_n, \kappa_n - d_{n+1})\end{aligned}$$

of a sound MMT $\mathcal{M}$, we define its set of *enabled timers*, denoted $\chi_0^{\mathcal{M}}(\rho)$, as

$$\chi_0^{\mathcal{M}}(\rho) = \{x \in X^{\mathcal{M}} \mid (\kappa_n - d_{n+1})(x) = 0\}.$$

The set of enabled timers of a run $\pi = q_0^{\mathcal{M}} \xrightarrow{i_1} q_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} q_n$, denoted $\chi_0^{\mathcal{M}}(\pi)$, is the union of all $\chi_0^{\mathcal{M}}(\rho)$ such that ρ is a timed run of $\mathcal{M}$ whose untimed projection is π.
Finally, the set of enabled timers of a state p, denoted $\chi_0^{\mathcal{M}}(p)$, is the union of all $\chi_0^{\mathcal{M}}(\pi)$ such that $\pi = q_0^{\mathcal{M}} \xrightarrow{w} p$ is a run of $\mathcal{M}$.

Example 9.2.10. Let us again consider the MMT $\mathcal{M}$ of Figure 9.1 and the timed run

$$\gamma = (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset) \xrightarrow{i} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0).$$

Then, $\chi_0^{\mathcal{M}}(\gamma) = \{x_1\}$. Furthermore,

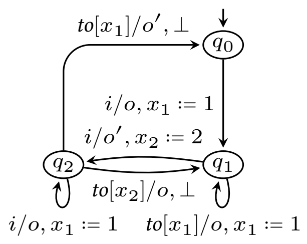
$$\chi_0^{\mathcal{M}}(q_0 \xrightarrow{i} q_1) = \chi_0^{\mathcal{M}}(\mathit{untime}(\gamma)) = \{x_1\}.$$

Finally, given the (untimed projections of the timed) runs of Examples 9.2.5 and 9.2.7, it is not hard to see that

$$\chi_0^{\mathcal{M}}(q_0) = \emptyset,$$
$$\chi_0^{\mathcal{M}}(q_1) = \{x_1\},$$

and

$$\chi_0^{\mathcal{M}}(q_2) = \{x_1, x_2\}.$$



This allows us to easily define when an MMT is complete: for every state, the set of defined transitions must be exactly those reading an input and those reading the timeouts of enabled timers.

Definition 9.2.11 (Complete MMT). An MMT $\mathcal{M}$ is said to be *complete* when for every state q of $\mathcal{M}$, $q \xrightarrow{i} \in \mathit{runs}(\mathcal{M})$ if and only if $i \in I \cup TO[\chi_0^{\mathcal{M}}(q)]$

The MMT of Figure 9.1 is complete.

Computing enabled timers

While computing the enabled timers of a timed run is easy (simply check the values of each timer in the last valuation), it is harder to determine the set for runs (and, thus, for states).

We first explain how to compute $\chi_0^{\mathcal{M}}(\pi)$ with $\pi \in \mathit{runs}(\mathcal{M})$. The idea is as follows. Whenever we take a $\mathit{to}[x]$-transition, there must be an earlier transition that (re)started x to some constant $c \in \mathbb{N}^{>0}$ (*i. e.*, the two transitions belong to the same block). Moreover, the elapsed time between the transition must be equal to c. Hence, we can accumulate some *constraints* along the transitions of π. Using these constraints, we are able to determine which timer can potentially time out at the end of π.

More formally, let

$$\pi = p_0 \xrightarrow[u_1]{i_1/o_1} p_1 \xrightarrow[u_2]{i_2/o_2} \cdots \xrightarrow[u_n]{i_n/o_n} p_n \in \mathit{runs}(\mathcal{M})$$

with $p_0 = q_0^{\mathcal{M}}$ (*i. e.*, we start from the initial state). If π is feasible, there must exist a timed run

$$\rho = (p_0, \emptyset) \xrightarrow{d_1} (p_0, \emptyset) \xrightarrow[u_1]{i_1/o_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots$$
$$\xrightarrow[u_n]{i_n/o_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1})$$

such that $\mathit{untime}(\rho) = \pi$ and

- for all $j \in \{1, \dots, n+1\}$, $d_j \in \mathbb{R}^{\geq 0}$,
- for any j and k such that $p_{j-1} \xrightarrow[(x,c)]{i_j} p_j \xrightarrow{i_{j+1}} \cdots \xrightarrow{i_k = \mathit{to}[x]} p_k$ is an x-spanning run,[6] the sum of the delays d_{j+1} to d_k must be equal to c, *i. e.*, $\sum_{\ell=j+1}^{k} d_\ell = c$, and
- for any j such that $u_j = (x, c)$ and there is no $k > j$ such that $i_k = \mathit{to}[x]$, then either x is restarted or stopped by some transition, or the run ends before $\mathit{to}[x]$ can occur or be processed.
  - In the first case, let $k > j$ such that $i_k \neq \mathit{to}[x]$ and $p_{k-1} \xrightarrow{i_k}$ restarts or stops x. Then, the sum of the delays d_{j+1} to d_k must be strictly less than c, *i. e.*, $\sum_{\ell=j+1}^{k} d_\ell \leq c$.
  - In the second case (so, $x \in \chi(p_n)$ and x does not time out after waiting d_{n+1}), the sum of the delays d_{j+1} to d_{n+1} must be strictly less than c, *i. e.*, $\sum_{\ell=j+1}^{n+1} d_\ell \leq c$.

6: That is, j and k belong to the same block.

Observe that these constraints are all linear. Moreover, if we consider the delays d_j as *variables*, one can still gather the constraints and use them to find a value for each d_j. We denote by $\mathrm{cnstr}(\pi)$ the set of constraints for π over the variables representing the delays. Notice that there may be multiple different solutions. Importantly, from the arguments given above, a solution always exists if and only if π is feasible.

Lemma 9.2.12. *Let $\mathcal{M}$ be a sound and complete MMT and $\pi \in \mathit{runs}(\mathcal{M})$. Then,* $\mathrm{cnstr}(\pi)$ *has a solution if and only if π is feasible.*

It remains to explain how to compute $\chi_0^{\mathcal{M}}(\pi)$ from $\mathrm{cnstr}(\pi)$. For every timer $x \in \chi(p_n)$ (*i. e.*, for every timer that may be enabled after reading π), we do as follows:

- Let $p_j \xrightarrow[(x,c)]{i_j} p_{j+1}$ be the last transition of π that (re)started x. Replace the constraint $\sum_{\ell=j+1}^{n+1} d_\ell \leq c$ by $\sum_{\ell=j+1}^{n+1} d_\ell = c$, *i. e.*, we force x to time out at the end of the timed run.
- Compute a solution to the refined $\mathrm{cnstr}(\pi)$. There are two possibilities:
  - A solution exists, meaning that there exists a timed run ending with
    $$(p_n, \kappa_n - d_{n+1}) \xrightarrow{\mathit{to}[x]} (p_{n+1}, \kappa_{n+1}) \xrightarrow{0} (p_{n+1}, \kappa_{n+1}).$$
    Hence, x is enabled in that timed run, meaning that it is also enabled for π, *i. e.*, $x \in \chi_0^{\mathcal{M}}(\pi)$.[7]
  - A solution does not exist, meaning that x can never time out after π, *i. e.*, $x \notin \chi_0^{\mathcal{M}}(\pi)$.

7: Observe that the untimed projection of the resulting timed run is no longer π, due to the new timeout action.

We can thus compute $\chi_0^{\mathcal{M}}(\pi)$ in finite time.

Lifting this procedure to any state q is complex, as a timer may only be enabled after looping over some states a precise number of times, *i. e.*, guessing which run to consider among all runs from $q_0^{\mathcal{M}}$ to q is a hard task. One way to achieve this is to compute the region automaton or the zone MMT, as introduced in Sections 9.5 and 9.7.1. Notice that, if there are only finitely many runs from $q_0^{\mathcal{M}}$ to q, it becomes easy to compute $\chi_0^{\mathcal{M}}(q)$: take the union of the enabled timers of each run. This idea will be used when learning (see the next chapter).

9.3. Relation with timed Mealy machine

In this section, we provide a construction to go from an MMT to a timed Mealy machine (TMM). That is, we convert timers into clocks. We also provide an example where the other direction (*i. e.*, from clocks into timers) does not hold. That is, we show the following theorem.[8]

8: Although we did not define $\mathit{toutputs}^{\mathcal{N}}(\cdot)$ for TMMs, its definition follows naturally from the semantics of $\mathcal{N}$.

Theorem 9.3.1. *For every MMT $\mathcal{M}$, there exists a TMM $\mathcal{N}$ such that $\mathit{toutputs}^{\mathcal{M}}(w) = \mathit{toutputs}^{\mathcal{N}}(w)$ for every timed word w over I. The opposite direction does not hold.*

Recall that $\Phi(C)$ denotes the set of *clock constraints* over the set of clocks C.[9]

9: See Section 9.2 for the definitions of clock constraints and TMMs.

9.3.1. From timers to clocks

Let $\mathcal{M}$ be an MMT. We assume that $X = \{x_1, \dots, x_n\}$ and construct a TMM $\mathcal{N}$ using X as its set of clocks (*i. e.*, $C = X$). The idea is as follows. Let V_{x_i} be

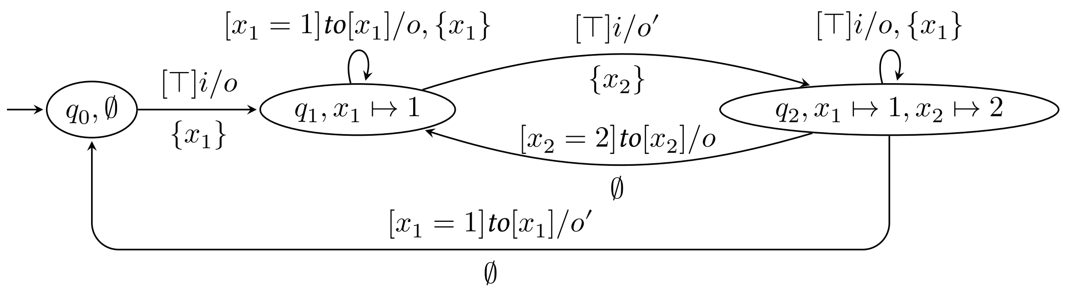

$[x_1 = 1]\mathit{to}[x_1]/o, \{x_1\}$
$[\top]i/o'$
$[\top]i/o, \{x_1\}$
$[\top]i/o$
$q_0, \emptyset$
$\{x_1\}$
$q_1, x_1 \mapsto 1$
$\{x_2\}$
$q_2, x_1 \mapsto 1, x_2 \mapsto 2$
$[x_2 = 2]\mathit{to}[x_2]/o$
$\emptyset$
$[x_1 = 1]\mathit{to}[x_1]/o'$
$\emptyset$

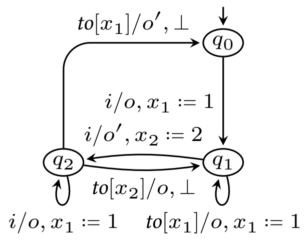
Figure 9.3: The timed Mealy machine constructed from the Mealy machine with timers of Figure 9.1, with $\mathit{Inv}((q_0, \emptyset)) = \top$, $\mathit{Inv}((q_1, x_1 \mapsto 1)) = 0 \leq x_1 \leq 1$, and $\mathit{Inv}((q_2, x_1 \mapsto 1, x_2 \mapsto 2)) = 0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 2$.

the set of all constants at which x_i is set on the transitions of $\mathcal{M}$, and V be the union of all V_x. Then, the states of $\mathcal{N}$ are pairs composed of a state q of $\mathcal{M}$, and a function $f : \chi(q) \to V$ with the intent that it stores the last value at which each timer was started.

While input-transitions can be triggered anytime, a $\mathit{to}[x_i]$-transition can only occur when the valuation of x_i is zero. Hence, in $\mathcal{N}$, the guard for a $\mathit{to}[x_i]$-transition is $x_i = c_i$. Finally, we use the invariants of the states of $\mathcal{N}$ to force a transition to be taken, *i.e.*, we set a limit over the time that can elapse within a state without triggering a transition.

We give a simple example before the formal definition. We then provide a more complex example, highlighting the interest of the functions f in the states of $\mathcal{N}$.

Example 9.3.2. Let $\mathcal{M}$ be the MMT of Figure 9.1 (which is repeated in the margin). Figure 9.3 gives the TMM $\mathcal{N}$ constructed from $\mathcal{M}$. We write $x_1 \mapsto 1, x_2 \mapsto 2$ to denote the function f such that $f(x_1) = 1$ and $f(x_2) = 2$. It is not hard to see that any timed run of $\mathcal{M}$ can be mimicked in $\mathcal{N}$, and vice-versa, given the invariants of the states and the guards of the transitions.


$\mathit{to}[x_1]/o', \bot$
q_0
$i/o, x_1 := 1$
$i/o', x_2 := 2$
q_2
q_1
$\mathit{to}[x_2]/o, \bot$
$i/o, x_1 := 1$
$\mathit{to}[x_1]/o, x_1 := 1$

Definition 9.3.3 (From MMT to TMM). Let $\mathcal{M} = (I, O, X, Q^{\mathcal{M}}, q_0^{\mathcal{M}}, \chi, \delta^{\mathcal{M}})$ be a sound and complete MMT. We define the TMM $\mathcal{N} = (I, O, X, Q^{\mathcal{N}}, q_0^{\mathcal{N}}, \mathit{Inv}, \delta^{\mathcal{N}})$ such that

- $Q^{\mathcal{N}} = \{(q, f) \mid q \in Q^{\mathcal{M}}, f : \chi(q) \to V\}$,
- $q_0^{\mathcal{N}} = (q_0, \emptyset)$,
- $\mathit{Inv} : Q^{\mathcal{N}} \to V$ is a total function defined as follows. For every $(q, f) \in Q^{\mathcal{N}}$, $\mathit{Inv}((q, f))$ ensures that the values of the clocks do not exceed $f(x)$, *i.e.*,

$$\mathit{Inv}((q, f)) = \begin{cases} \top & \text{if } \chi(q) = \emptyset \\ \bigwedge_{x \in \chi(q)} 0 \leq x \leq f(x) & \text{otherwise.} \end{cases}$$

- The transition relation $\delta^{\mathcal{N}}$ is defined by copying the transitions of $\mathcal{M}$ and "updating" the function f according to the updates. That is, for every $(q, f) \in Q^{\mathcal{N}}$ and transition $q \xrightarrow[u]{i/o} q'$ of $\mathcal{M}$, we define the transition

$$(q, f) \xrightarrow[r]{[g]i/o} (q', f') \in \mathit{runs}(\mathcal{N})$$

with

$$r = \begin{cases} \{x\} & \text{if } u = (x, \cdot) \\ \emptyset & \text{otherwise,} \end{cases}$$

$$g = \begin{cases} x = f(x) & \text{if } i = \mathit{to}[x] \\ \top & \text{otherwise,} \end{cases}$$

and $f' : \chi(q') \to V$ is such that

$$\forall x \in \chi(q') : f'(x) = \begin{cases} c & \text{if } u = (x, c) \\ f(x) & \text{if } u \neq (x, \cdot). \end{cases}$$

Let us now give a more complex example, illustrating the need of the functions in the states of the TMM.

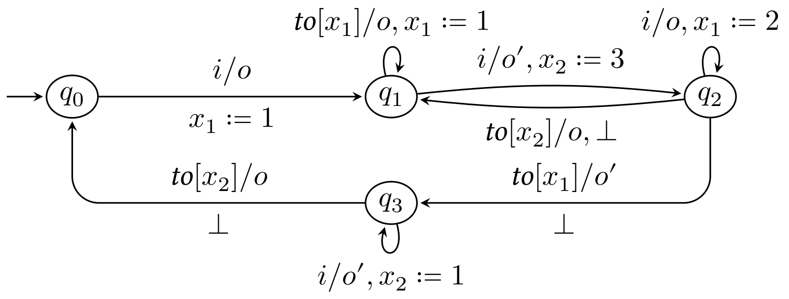
Example 9.3.4. Let us consider the MMT of Figure 9.4a. Observe that the self-loops over q_2 and q_3 restart x_1 and x_2 (respectively) with values that are different from the first time each timer is started. Hence, in the corresponding TMM, states q_2 and q_3 need to be split, depending on whether the self-loops have already been triggered. Indeed, the conditions to remain in the state, and when a timeout can occur must change. Figure 9.4b gives the TMM. Again, it is not hard to see any timed run of the MMT can be reproduced in the TMM, and vice-versa.

Observe that, in the worst case, Definition 9.3.3 induces an exponential blowup in the number of timers. Indeed, the number of states of $\mathcal{N}$ is bounded by $|Q^{\mathcal{M}}| \cdot |V|^{|X|}$. We now give an example where an exponential number of states is required.

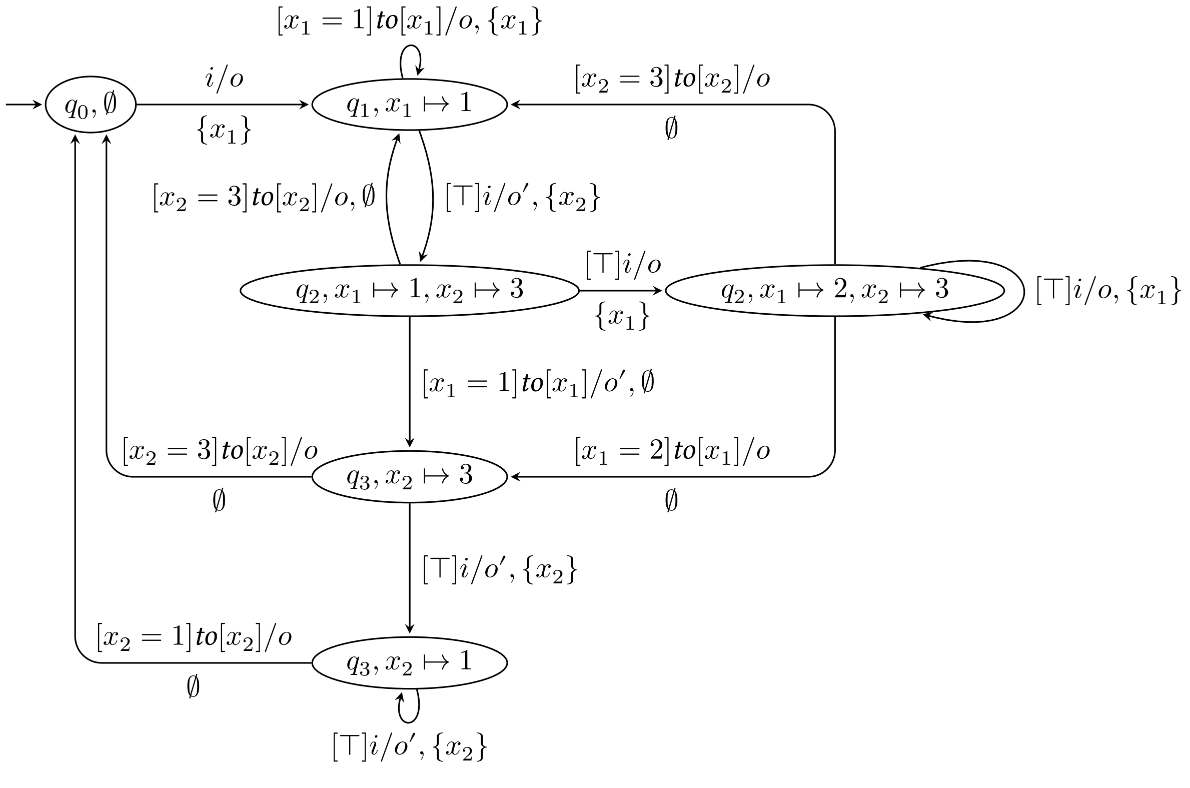
Example 9.3.5. Let $n \in \mathbb{N}^{>0}$, $I = \{i_1, i_2, j\}$, and $X = \{x_1, \dots, x_n\}$. Let $\mathcal{M}$ be the MMT of Figure 9.5. For clarity, outputs are not represented on the figure, and any missing transition leads to q_0. We have $V_{x_1} = \dots = V_{x_n} = V = \{1, 2\}$. Let us focus on the states p_1 to p_n. Notice that, for every $i, j \in \{1, \dots, n\}$, it is possible to reach the state p_i such that the timer x_j was last started at 1 (resp. 2). By Definition 9.3.3, each p_i then induces 2^n states in the TMM $\mathcal{N}$. That is, $|Q^{\mathcal{N}}|$ is exponential in the number of timers of $\mathcal{M}$.

In conclusion, since an MMT can be converted into a TMM, we can leverage many properties and algorithms developed for timed automata (see Chapter 8).[10] Nonetheless, in the next sections, we develop a more specific theory for MMTs. That is, we adapt the notion of regions and zones and show that reachability remains PSPACE-complete. Furthermore, we study whether it is possible to observe every untimed behavior of an MMT with timed runs in which all delays are positive.

10: However, due to the exponential blowup of Definition 9.3.3, complexity results may not transfer directly.



(a) The MMT with $\chi(q_0) = \emptyset$, $\chi(q_1) = \{x_1\}$, $\chi(q_2) = \{x_1, x_2\}$, and $\chi(q_3) = \{x_2\}$.


$q_0, \emptyset$
$q_1, x_1 \mapsto 1$
$q_2, x_1 \mapsto 1, x_2 \mapsto 3$
$q_2, x_1 \mapsto 2, x_2 \mapsto 3$
$q_3, x_2 \mapsto 3$
$q_3, x_2 \mapsto 1$
$[x_1 = 1]\mathit{to}[x_1]/o, \{x_1\}$
i/o
$\{x_1\}$
$[x_2 = 3]\mathit{to}[x_2]/o$
$\emptyset$
$[x_2 = 3]\mathit{to}[x_2]/o, \emptyset$
$[\top]i/o', \{x_2\}$
$[\top]i/o$
$\{x_1\}$
$[\top]i/o, \{x_1\}$
$[x_1 = 1]\mathit{to}[x_1]/o', \emptyset$
$[x_2 = 3]\mathit{to}[x_2]/o$
$\emptyset$
$[x_1 = 2]\mathit{to}[x_1]/o$
$\emptyset$
$[\top]i/o', \{x_2\}$
$[x_2 = 1]\mathit{to}[x_2]/o$
$\emptyset$
$[\top]i/o', \{x_2\}$

(b) The TMM with $\mathit{Inv}((q_0, \emptyset)) = \top$, $\mathit{Inv}((q_1, x_1 \mapsto 1)) = 0 \leq x_1 \leq 1$, $\mathit{Inv}((q_2, x_1 \mapsto 1, x_2 \mapsto 3)) = 0 \leq x_1 \leq 1 \land 0 \leq x_3 \leq 3$, $\mathit{Inv}((q_2, x_1 \mapsto 2, x_2 \mapsto 3)) = 0 \leq x_1 \leq 2 \land 0 \leq x_3 \leq 3$, $\mathit{Inv}((q_2, x_2 \mapsto 3)) = 0 \leq x_3 \leq 3$, and $\mathit{Inv}((q_2, x_2 \mapsto 1)) = 0 \leq x_3 \leq 1$.

Figure 9.4: A Mealy machine with timers and its constructed timed Mealy machine.

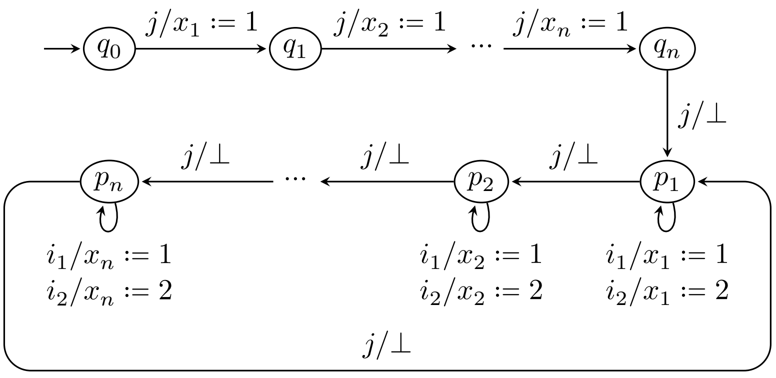
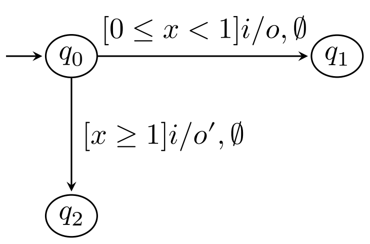
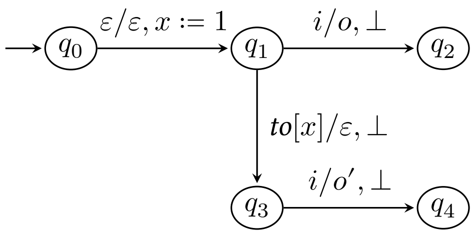


Figure 9.5: An MMT, with $\chi(q_0) = \emptyset, \chi(q_1) = \{x_1\}, \dots, \chi(q_n) = \{x_1, \dots, x_n\} = \chi(p_1) = \dots = \chi(p_n)$, for which Definition 9.3.3 yields an exponential number of states. For clarity, outputs are omitted and any missing transition goes to q_0.



(a) The TMM with $Inv(q_0) = Inv(q_1) = Inv(q_2) = \top$.



(b) The MMT with $\chi(q_0) = \chi(q_2) = \chi(q_3) = \chi(q_4) = \emptyset$, and $\chi(q_1) = \{x\}$.

Figure 9.6: A timed Mealy machine and a candidate corresponding Mealy machine with timers.

9.3.2. From clocks to timers

Let us now argue that it is not always possible to convert a TMM into an MMT. Since clocks are always running, it is possible to describe transitions from the initial state that can only occur with a specific timing in a TMM. However, these timed behaviors cannot be reproduced in an MMT without changing the definition (as no timer is active in the initial state). Hence, we allow in this section the use of ε-transitions in an MMT. That is, timers can be started without having to read any symbol. Furthermore, we assume that timeout transitions do not output any symbol (*i. e.*, only the input transitions can produce an output). Still, there exists a TMM that cannot be perfectly simulated by any MMT.

Let $\mathcal{N}$ be the TMM of Figure 9.6a. Let $w = 1 \cdot i \cdot 1$ be a timed word, and ρ be the timed run of $\mathcal{N}$ reading w, *i. e.*,

$$\rho = (q_0, x = 0) \xrightarrow{1} (q_0, x = 1) \xrightarrow{i/o'} (q_2, x = 1) \xrightarrow{1} (q_2, x = 2).$$

Observe that it is the unique run of $\mathcal{N}$ that can read w, as there is a single guard that is satisfied by the valuation $x = 1$. That is, $\mathit{toutputs}^{\mathcal{N}}(w) = 1 \cdot o' \cdot 1$.

If we were to create a corresponding MMT $\mathcal{M}$, we would need a timer, say y, that mimics x and that is used to know whether we should take the transition $q_0 \xrightarrow{i} q_1$ or $q_0 \xrightarrow{1} q_2$. More specifically, y should be started by a transition reading ε (*i. e.*, reading no symbol) and at value 1 (given the constant appearing in the guards of $\mathcal{N}$).

Figure 9.6b gives a candidate MMT following these ideas. Let us now look at two possible timed runs reading w:

$$\sigma = (q_0, \emptyset) \xrightarrow{0} (q_0, \emptyset) \xrightarrow{\varepsilon} (q_1, x = 1) \xrightarrow{1} (q_1, x = 0) \xrightarrow{i/o} (q_2, \emptyset) \xrightarrow{1} (q_2, \emptyset)$$

and

$$\begin{aligned}\sigma' = (q_0, \emptyset) &\xrightarrow{0} (q_0, \emptyset) \xrightarrow{\varepsilon} (q_1, x = 1) \xrightarrow{1} (q_1, x = 0) \xrightarrow{to[x]/\varepsilon} (q_3, \emptyset) \\ &\xrightarrow{0} (q_3, \emptyset) \xrightarrow{i/o'} (q_4, \emptyset) \xrightarrow{1} (q_4, \emptyset).\end{aligned}$$

Hence, $\mathit{toutputs}^{\mathcal{M}}(w)$ contains (at least) $1 \cdot o \cdot 1$ and $1 \cdot o' \cdot 1$. Since the first output word is not in $\mathit{toutputs}^{\mathcal{N}}(w)$, we conclude that $\mathcal{M}$ is not a valid candidate for the simulation of $\mathcal{N}$. However, it is not hard to see that any MMT that does not follow the above ideas cannot simulate $\mathcal{N}$. We thus conclude that there are TMMs that cannot be reproduced by MMTs, even by changing the syntax and semantics of MMTs.

9.4. Equivalence of two Mealy machines with timers

Given two sound and complete MMTs, it may be useful to decide whether they describe the same timed behaviors, despite the fact that they do not use the same timers. In this section, we present two different approaches to test equivalence: the first one "hides" the timeouts behind the delays (so, in a timed context), while the second one*symbolically* describes (untimed) runs. We also claim that the second implies the first.

9.4.1. Timed equivalence

For the timed equivalence approach, recall that it is not possible to let time elapse when a timer times out. Hence, a complete MMT $\mathcal{M}$ can be assumed to automatically process timeouts when they occur. From a timed run

$$\begin{aligned}\rho = (p_0, \kappa_0) &\xrightarrow{d_1} (p_0, \kappa_0 - d_1) \xrightarrow{i_1/o_1} \cdots \\ &\xrightarrow{d_n} (p_{n-1}, \kappa_{n-1} - d_n) \xrightarrow{i_n/o_n} (p_n, \kappa_n) \\ &\xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1}),\end{aligned}$$

we construct a timed word over I (*i. e.*, without any timeout actions) as follows:

- Drop every timeout transition from $d_1 \cdot i_1 \cdots i_n \cdot d_{n+1}$.
- The resulting word may have two consecutive delays d_i and d_{i+1} without any action in between. In that case, simply sum d_i and d_{i+1} to obtain a new delay. Repeat this until we obtain a well-formed timed word over I.

We call such a word a *timed input word* (*tiw*, for short) and denote by $\mathit{tiw}(\rho)$ the tiw obtained from the timed run ρ. It is not hard to reconstruct a timed run from a tiw: simply follow the provided delays and input symbols, while automatically adding timeout actions whenever needed. As more than one timer may time out simultaneously (or some timers time out at the same time an input must be processed), a single tiw w can induce many timed runs. Let *tiwruns*(w) be the set of all timed runs induced by w.

Recall that any timed run ρ yields a timed word *tow*(ρ) over O, called a timed output word (tow, in short). We highlight that the number of output symbols in *tow*(ρ) may be greater than the number of input symbols in *tiw*(ρ), due to the timeouts. We lift *toutputs*(w), with w a tiw, to denote the set of all tows produced by the timed runs in *tiwruns*(w).

Example 9.4.1. Let $\mathcal{M}$ be the MMT of Figure 9.1 (repeated in the margin) and $w = 1 \cdot i \cdot 1 \cdot i \cdot 1.5$ be a tiw. There are two possible timed runs in *tiwruns*(w):[11]

$$\begin{aligned}\rho = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 0, x_2 = 2) \xrightarrow{0} (q_2, x_1 = 0, x_2 = 2)\\ &\xrightarrow[\perp]{\mathit{to}[x_1]/o'} (q_0, \emptyset) \xrightarrow{1.5} (q_0, \emptyset)\end{aligned}$$

and

$$\begin{aligned}\sigma = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\ &\xrightarrow[(x_1,1)]{\mathit{to}[x_1]/o} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 1, x_2 = 2) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1)\\ &\xrightarrow[\perp]{\mathit{to}[x_1]/o'} (q_0, \emptyset) \xrightarrow{0.5} (q_0, \emptyset).\end{aligned}$$

Observe that we insert the timeouts of x_1 and x_2 whenever needed. We thus have two different tows:

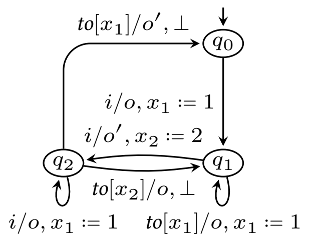
$$\mathit{tow}(\rho) = 1 \cdot o \cdot 1 \cdot o' \cdot 0 \cdot o' \cdot 1.5$$

and

$$\mathit{tow}(\sigma) = 1 \cdot o \cdot 1 \cdot o \cdot 0 \cdot o' \cdot 1 \cdot o' \cdot 0.5.$$

Hence, $|\mathit{toutputs}(w)| = 2$.

11: These are the timed runs ρ and σ from Examples 9.2.5 and 9.2.7.



$\mathit{to}[x_1]/o', \perp$
q_0
$i/o, x_1 := 1$
$i/o', x_2 := 2$
q_2
q_1
$\mathit{to}[x_2]/o, \perp$
$i/o, x_1 := 1$
$\mathit{to}[x_1]/o, x_1 := 1$

We can now easily define the equivalence of two MMTs $\mathcal{M}$ and $\mathcal{N}$: for every tiw, both $\mathcal{M}$ and $\mathcal{N}$ produce the same tows.

Definition 9.4.2 (Timed equivalence)**.** Two sound and complete MMTs $\mathcal{M}$ and $\mathcal{N}$ are *timed equivalent*, denoted by $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$, if and only if *toutputs*$^{\mathcal{M}}(w) =$ *toutputs*$^{\mathcal{N}}(w)$ for all tiws w.

9.4.2. Symbolic equivalence

The second approach aims at abstracting the timeouts while remaining in the "untimed world". For this purpose, we define *symbolic words* as words $\mathtt{w}$ over the alphabet $\mathtt{A} = I \cup \mathit{TO}[\mathbb{N}^{>0}]$ to describe the x-spanning sub-runs of a run in the following way. Along a run π of a sound MMT, for any $\mathit{to}[x]$-transition there must exist an earlier transition (re)starting x. The part of the run between the last such transition and the $\mathit{to}[x]$ is x-spanning. Hence, for a given $\mathit{to}[x]$-transition of π, there exists a unique transition that is the source of this timeout transition. Let $w = i_1 \cdots i_n$ be a word over $A(\mathcal{M})$ that is the label of a run

$$\pi = p_0 \xrightarrow[u_1]{i_1} p_1 \xrightarrow[u_2]{i_2} \cdots \xrightarrow[u_n]{i_n} p_n \in \mathit{runs}(\mathcal{M}).$$

The *symbolic word* (*sw*, for short) *of* w is the word $\overline{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ over $\mathtt{A}$ such that, for every $k \in \{1, \dots, n\}$,

- ▶ $\mathtt{i}_k = i_k$ if $i_k \in I$, and
- ▶ $\mathtt{i}_k = \mathit{to}[j]$ where $j < k$ is the index of the last transition (re)starting x if $i_k = \mathit{to}[x]$.

Conversely, given a symbolic word $\mathtt{w} = \mathtt{i}_1 \dots \mathtt{i}_n$ over $\mathtt{A}$, one can convert it into a run $q_0 \xrightarrow{w}$ using concrete timeout symbols such that $\overline{w} = \mathtt{w}$ if such a run exists in $\mathcal{M}$. In short, for a symbol $\mathit{to}[j]$, it suffices to retrieve the j-th transition of the run. If that transition (re)starts a timer x, $\mathit{to}[j]$ is then replaced by $\mathit{to}[x]$. Section C.1 gives further details.

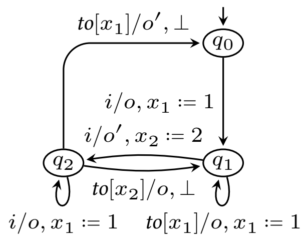
Example 9.4.3. Let $\mathcal{M}$ be the MMT of Figure 9.1 (repeated in the margin) and

$$\pi = q_0 \xrightarrow[(x_1,1)]{i} q_1 \xrightarrow[(x_1,1)]{\mathit{to}[x_1]} q_1 \xrightarrow[(x_1,1)]{\mathit{to}[x_1]} q_1 \in \mathit{runs}(\mathcal{M}).$$

Let us construct the symbolic word $\mathtt{w} = \mathtt{i}_1 \cdot \mathtt{i}_2 \cdot \mathtt{i}_3$ such that $\overline{i \cdot \mathit{to}[x_1] \cdot \mathit{to}[x_1]} = \mathtt{w}$. As the first action of π is the input i, we get $\mathtt{i}_1 = i$. The second action of π is $\mathit{to}[x_1]$ and the last transition to (re)start x_1 is the first transition of π. So, $\mathtt{i}_2 = \mathit{to}[1]$. Likewise, the last symbol $\mathtt{i}_3$ of $\mathtt{w}$ must be $\mathit{to}[2]$, as the second transition of π restarts x_1. Hence, $\mathtt{w} = i \cdot \mathit{to}[1] \cdot \mathit{to}[2]$.
In the opposite direction, it is not hard to see that the symbolic word $\mathtt{w} = i \cdot i \cdot \mathit{to}[2] \cdot \mathit{to}[1]$ induces the run

$$q_0 \xrightarrow[(x_1,1)]{i} q_1 \xrightarrow[(x_2,2)]{i} q_2 \xrightarrow{\mathit{to}[x_2]} q_2 \xrightarrow{\mathit{to}[x_1]} q_0 \in \mathit{runs}(\mathcal{M})$$

such that $\overline{i \cdot i \cdot \mathit{to}[x_2] \cdot \mathit{to}[x_1]} = \mathtt{w}$.



We now define a notion of *symbolic equivalence* between two MMTs $\mathcal{M}$ and $\mathcal{N}$ such that for any symbolic word $\mathtt{w}$, $\mathcal{M}$ has a feasible run reading $\mathtt{w}$ if and only if $\mathcal{N}$ has a feasible run reading $\mathtt{w}$, and outputs and updates starting spanning sub-runs are the same (up to timer renaming).

Definition 9.4.4 (Symbolic equivalence). Two sound and complete MMTs $\mathcal{M}$ and $\mathcal{N}$ are *symbolically equivalent*, denoted by $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, if for every symbolic word $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ over $\mathtt{A}$,

$$q_0^{\mathcal{M}} \xrightarrow[u_1]{\mathtt{i}_1/o_1} q_1 \xrightarrow[u_2]{\mathtt{i}_2/o_2} \cdots \xrightarrow[u_n]{\mathtt{i}_n/o_n} q_n$$

is a feasible run in $\mathcal{M}$ (where every q_j is a state of $\mathcal{M}$) if and only if

$$q_0^{\mathcal{N}} \xrightarrow[u'_1]{\mathtt{i}_1/o'_1} q'_1 \xrightarrow[u'_2]{\mathtt{i}_2/o'_2} \cdots \xrightarrow[u'_n]{\mathtt{i}_n/o'_n} q'_n$$

is feasible in $\mathcal{N}$ (where every q'_j is a state of $\mathcal{N}$). Moreover,

- $o_j = o'_j$ for all $j \in \{1, \dots, n\}$, and
- if $q_{j-1} \xrightarrow{\mathtt{i}_j \cdots \mathtt{i}_k} q_k$ is spanning, then $u_j = (x, c)$, $u'_j = (x', c')$, and $c = c'$.

Notice that the run $q_{j-1} \xrightarrow{\mathtt{i}_j \cdots \mathtt{i}_k} q_k$ is spanning in $\mathcal{M}$ if and only if the run $q'_{j-1} \xrightarrow{\mathtt{i}_j \cdots \mathtt{i}_k} q'_k$ is spanning in $\mathcal{N}$ as both machines read the same symbolic word. Notice also that no condition is imposed on the updates u_j, u'_j appearing outside the start of spanning runs. As outputs and updates at the start of spanning runs are the same, symbolic equivalence implies timed equivalence. Section C.2 gives a proof and a counterexample for the reverse implication.

Proposition 9.4.5. *Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.*

9.5. Reachability and regions

The *reachability problem* asks, given an MMT $\mathcal{M}$ and a state q, whether there exists a timed run $\rho \in \mathit{truns}(\mathcal{M})$ from the initial configuration $(q_0^{\mathcal{M}}, \emptyset)$ to some configuration (q, κ). In this section, we argue that this problem is PSPACE-complete. For simplicity, we ignore all outputs.

Theorem 9.5.1. *The reachability problem for MMTs is* PSPACE*-complete.*

For hardness, we reduce from the acceptance problem for *linear bounded Turing machines* (*LBTM*, for short), as done for timed automata (see Section 8.3). In short, given an LBTM $\mathcal{A}$ and a word w of length n, we construct an MMT that uses n timers x_i, $1 \leq i \leq n$, such that the timer x_i encodes the value of the i-th cell of the tape of $\mathcal{A}$. We also rely on a timer x that is always (re)started at 1, and is used to synchronize the x_i timers and the simulation of $\mathcal{A}$. The simulation is split into *phases*: the MMT first seeks the symbol on the current cell i of the tape (which can be derived from the moment at which the timer x_i times out, using the number of times x timed out since the beginning of the phase). Then, the MMT simulates a transition of $\mathcal{A}$ by restarting x_i, reflecting the new value of the i-th cell. Finally, the MMT can reach a designated state if and only if $\mathcal{A}$ is in an accepting state. Therefore, the reachability problem is PSPACE-hard. Complete details are provided in Section C.3

For membership, we follow the classical argument used to establish that the reachability problem for timed automata is in PSPACE (see Section 8.3): we first define *region automata* for MMTs (which are a simplification of region automata for timed automata) and observe that reachability in an MMT reduces to reachability in the corresponding region automaton. The region automaton is of size exponential in the number of timers and polynomial in the number of states of the MMT. Hence, the reachability problem for MMTs is in PSPACE via standard arguments.

We define region automata for MMTs much like they are defined for timed automata [Alu99; AD94; BK08] (see Definition 8.3.2).

[Alu99]: Alur (1999), "Timed Automata"
[AD94]: Alur et al. (1994), "A Theory of Timed Automata"
[BK08]: Baier et al. (2008), *Principles of model checking*

Definition 9.5.2 (Timer region). Let $\mathcal{M} = (I, O, X, Q, q_0, \chi, \delta)$ be an MMT. Two valuations κ and κ' are *timer-equivalent*, denoted by $\kappa \cong \kappa'$, if $\mathsf{dom}(\kappa) = \mathsf{dom}(\kappa')$ and the following hold:

- for all $x \in X$, $\lfloor \kappa(x) \rfloor = \lfloor \kappa'(x) \rfloor$ and
- for all $x \in X$, $\mathsf{frac}(\kappa(x)) = 0$ if and only if $\mathsf{frac}(\kappa'(x)) = 0$, and
- for all $x_1, x_2 \in X$, $\mathsf{frac}(\kappa(x_1)) \leq \mathsf{frac}(\kappa(x_2))$ if and only if $\mathsf{frac}(\kappa'(x_1)) \leq \mathsf{frac}(\kappa'(x_2))$.

A *timer region* for $\mathcal{M}$ is an equivalence class of timer valuations induced by $\cong$. We lift the relation to configurations: $(q, \kappa) \cong (q', \kappa')$ if and only if $\kappa \cong \kappa'$ and $q = q'$.

We are now able to define an automaton from $\cong$.[13] We introduce a new symbol τ that is used to abstract the non-zero delays. That is, every delay transition $q \xrightarrow{d} p$ with $d > 0$ is replaced by some transition reading τ.

13: As we are solely interested in encoding the runs of an MMT (where outputs are ignored), we do not define the set of final states. It can be defined as being exactly the set of states of the automaton.

Definition 9.5.3 (Region automaton). The *region automaton* of $\mathcal{M}$ is denoted $\mathcal{R}(\mathcal{M})$ and such that

- its alphabet is $\Sigma = \{\tau\} \cup A(\mathcal{M})$,
- its set of states $Q^{\mathcal{R}(\mathcal{M})}$ is the quotient of the configurations by $\cong$, *i.e.*

$$Q^{\mathcal{R}(\mathcal{M})} = \{(q, \kappa) \mid q \in Q, \kappa \in \mathsf{Val}(\chi(q))\}_{/\cong},$$

- its initial state $q_0^{\mathcal{R}(\mathcal{M})}$ is the class of the initial configuration of $\mathcal{M}$, *i.e.*,

$$q_0^{\mathcal{R}(\mathcal{M})} = [\![(q_0^{\mathcal{M}}, \emptyset)]\!]_{\cong} = (q_0^{\mathcal{M}}, [\![\emptyset]\!]_{\cong})$$

  (by definition of $\cong$),
- its transition relation $\delta \subseteq S \times \Sigma \times S$ includes
  - $[\![(q, \kappa)]\!]_{\cong} \xrightarrow{\tau} [\![(q, \kappa - d)]\!]_{\cong}$ if $(q, \kappa) \xrightarrow{d} (q, \kappa - d)$ in $\mathcal{M}$ whenever $d > 0$, and
  - $[\![(q, \kappa)]\!]_{\cong} \xrightarrow{i} [\![(q', \kappa')]\!]_{\cong}$ if $(q, \kappa) \xrightarrow[u]{i} (q', \kappa')$ in $\mathcal{M}$.

Definition 8.3.3. A relation R on the states $Q^{\mathcal{M}}$ of $\mathcal{M}$ is a *time-abstracted bisimulation* if, for every $i \in I$, and configurations (q_1, κ_1) and (q_2, κ_2),

- $(q_1, \kappa_1) \mathrel{R} (q_2, \kappa_2)$ and
- $(q_1, \kappa_1) \xrightarrow{d_1 \cdot i} (q_1', \kappa_1')$ for some delay $d_1 \in \mathbb{R}^{\geq 0}$

imply

- $(q_2, \kappa_2) \xrightarrow{d_2 \cdot i} (q_2', \kappa_2')$ for some $d_2 \in \mathbb{R}^{\geq 0}$ and
- $(q_1', \kappa_1') \mathrel{R} (q_2', \kappa_2')$,

and vice-versa.

It is easy to check that the timer-equivalence relation on configurations is a *(strong) time-abstracting bisimulation* [TY01; Cla+18] (see Definition 8.3.3). That is, for all $(q_1, \kappa_1) \cong (q_2, \kappa_2)$ the following holds:

- if $(q_1, \kappa_1) \xrightarrow[u]{i} (q_1', \kappa_1')$, then there is $(q_2, \kappa_2) \xrightarrow[u]{i} (q_2', \kappa_2')$ with $(q_1', \kappa_1') \cong (q_2', \kappa_2')$,

[TY01]: Tripakis et al. (2001), "Analysis of Timed Systems Using Time-Abstracting Bisimulations"
[Cla+18]: Clarke et al. (2018), *Handbook of Model Checking*

- if $(q_1, \kappa_1) \xrightarrow{d_1} (q_1, \kappa_1')$, then there exists $(q_2, \kappa_2) \xrightarrow{d_2} (q_2, \kappa_2')$ where d_1, $d_2 > 0$ may differ such that $(q_1, \kappa_1') \cong (q_2, \kappa_2')$, and
- the above also holds if (q_1, κ_1) and (q_2, κ_2) are swapped.

Using this property, we can prove the following about $\mathcal{R}(\mathcal{M})$:

- $\mathcal{R}(\mathcal{M})$ is finite, by a bound over the number of states, which is polynomial in $|Q^{\mathcal{M}}|$ and exponential in $|X|$, and
- there exists a timed run in $\mathcal{M}$ from (q, κ) to (q', κ') if and only if there exists a run from the class of (q, κ) to the class of (q', κ') in $\mathcal{R}(\mathcal{M})$, *i.e.*, there is an equivalence between the runs of both machines.[14]

14: Again, we ignore outputs.

Lemma 9.5.4. *Let $\mathcal{M}$ be an MMT and $\mathcal{R}(\mathcal{M})$ be its region automaton. For a timer $x \in X$, c_x denotes the largest constant to which x is updated in $\mathcal{M}$. Let $C = \max_{x \in X} c_x$. Then, the number of states of $\mathcal{R}(\mathcal{M})$ is bounded by*

$$|Q^{\mathcal{M}}| \cdot |X|! \cdot 2^{|X|} \cdot (C+1)^{|X|}.$$

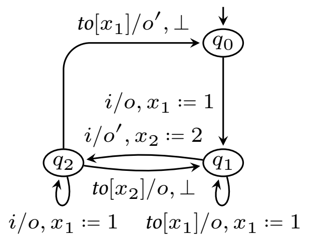
Moreover, for all $q, q' \in Q^{\mathcal{M}}, \kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q)), \kappa' \in \mathsf{Val}(\chi^{\mathcal{M}}(q'))$, there exists a timed run from (q, κ) to (q', κ') in $\mathcal{M}$ if and only if there exists a run from $[\![(q, \kappa)]\!]_{\cong}$ to $[\![(q', \kappa')]\!]_{\cong}$ in $\mathcal{R}(\mathcal{M})$.

Proof. For the first statement of the lemma, recall that each state of the region automaton is of the form $(q, [\![\kappa]\!]_{\cong})$ with $q \in Q^{\mathcal{M}}$. So, the number of possible q is equal to $|Q^{\mathcal{M}}|$. Concerning the number of region classes $[\![\kappa]\!]_{\cong}$, $(C+1)^{|X|}$ is related to the integer parts of the timers between 0 and C, $2^{|X|}$ to which timers have a zero fractional part, and $|X|!$ to the order of these fractional parts.
The second statement of the lemma follows from the definition of the region automaton $\mathcal{R}(\mathcal{M})$. Notice that delay transitions with delay 0 in $\mathcal{M}$ disappear in $\mathcal{R}(\mathcal{M})$. □

Despite its exponential size, it is possible to compute the region automaton on the fly. That is, seeking a run traversing a state q can be done in PSPACE. This is formalized after the example.

Example 9.5.5. Let us consider the MMT $\mathcal{M}$ of Figure 9.1 (repeated in the margin) and the timed run π given in Example 9.2.7:

$$\begin{aligned}\pi = (q_0, \emptyset) &\xrightarrow{0.5} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 1, x_2 = 2) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1)\\ &\xrightarrow[(x_1,1)]{i/o} (q_2, x_1 = 1, x_2 = 1) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 0)\\ &\xrightarrow[\bot]{to[x_2]/o} (q_1, x_1 = 0) \xrightarrow{0} (q_1, x_1 = 0)\\ &\xrightarrow[(x_1,1)]{to[x_1]/o} (q_1, x_1 = 1) \xrightarrow{0.5} (q_1, x_1 = 0.5).\end{aligned}$$



The corresponding run π' in the region automaton $\mathcal{R}(\mathcal{M})$ is

$$\begin{aligned}\pi' = (q_0, [\![\emptyset]\!]_{\cong}) &\xrightarrow{\tau} (q_0, [\![\emptyset]\!]_{\cong}) \xrightarrow{i} (q_1, [\![x_1 = 1]\!]_{\cong})\\ &\xrightarrow{i} (q_2, [\![x_1 = 1, x_2 = 2]\!]_{\cong}) \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 1]\!]_{\cong})\\ &\xrightarrow{i} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}) \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 0]\!]_{\cong})\\ &\xrightarrow{to[x_2]} (q_1, [\![x_1 = 0]\!]_{\cong}) \xrightarrow{to[x_1]} (q_1, [\![x_1 = 1]\!]_{\cong})\\ &\xrightarrow{\tau} (q_1, [\![0 < x_1 < 1]\!]_{\cong}).\end{aligned}$$

Notice that the transitions with delay zero of π do not appear in π'.

We can now prove the upper bound (*i.e.*, the PSPACE membership) of Theorem 9.5.1.

Upper bound of Theorem 9.5.1. To decide the reachability problem for MMTs, by Lemma 9.5.4, we can simulate a run of the corresponding region automaton. Instead of constructing the region automaton in full, we can do so "on the fly". This yields a nondeterministic decision procedure for the reachability problem which, due to the form $(q, [\![\kappa]\!]_{\cong})$ of the states of $\mathcal{R}(\mathcal{M})$, requires polynomial space only. Since NPSPACE $=$ PSPACE, we obtain the upper bound stated in Theorem 9.5.1. □

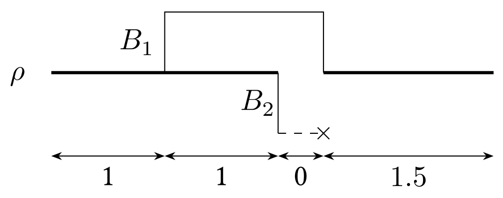
9.6. Non-determinism due to zero delays

In a timed run

$$\begin{aligned}\rho = (p_0, \kappa_0) \xrightarrow{d_1} (p_0, \kappa_0 - d_1) &\xrightarrow[u_1]{i_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots\\ &\xrightarrow[u_n]{i_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1})\end{aligned}$$

of an MMT $\mathcal{M}$, it may happen that some $d_j = 0$, *i.e.*, that two actions occur at the same time. As already said, $\mathcal{M}$ can process i_j and i_{j+1} in any order, meaning that the behavior of $\mathcal{M}$ is non-deterministic (see Example 9.2.7). While this non-determinism can be avoided by imposing that any delay is positive, doing so may prevent some feasible untimed runs to be observed via timed runs. That is, there may be a feasible untimed run of $\mathcal{M}$ that requires a zero delay in any corresponding timed run. For instance, it is the case for the untimed projection of the timed run π of Example 9.2.7.

In this section, we study the problem of deciding whether an MMT has some untimed runs requiring zero delays. Our main tool will be to *wiggle* the blocks in a run, in the sense that we will slightly change the delays before a block in order to remove these zero delays, whenever possible, while still seeing the same actions *in the same order*. In order to be able to do so, we require that we have non-zero delays at the very start and at the very end of ρ, *i.e.*, that $d_1, d_{n+1} > 0$. Furthermore, we require that none of the active timers in q_n can timeout in the last configuration, *i.e.*, that $(\kappa_n - d_{n+1})(x) > 0$ for every



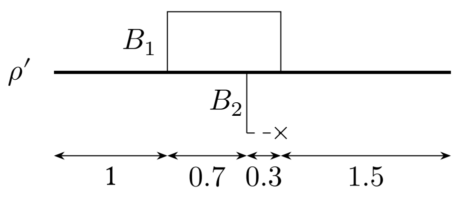


Figure 9.7: Modifying the delays in order to remove a race.

$x \in \chi(q_n)$. Such runs are called *padded*, and we denote by *ptruns*$(\mathcal{M})$ the set of all padded timed runs of $\mathcal{M}$.

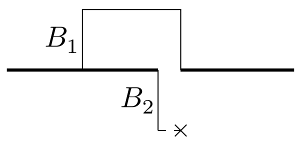
In this section, we solely focus on padded runs (which will allow to wiggle the blocks that appear at the start or at the end of a run). Let us first properly introduce the decision problem we study. We call a situation where two actions occur at the same time a *race*.

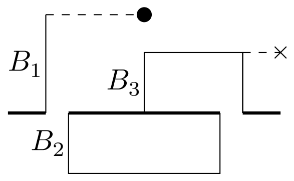
Definition 9.6.1 (Race)**.** Let B, B' be two blocks of a padded timed run ρ with timer fates γ and γ'. We say that B and B' *participate in a race* if:

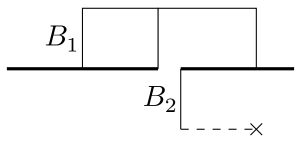
- either there exist actions $i \in B$ and $i' \in B'$ such that the sum of the delays between i and i' in ρ is equal to zero, *i.e.*, no time elapses between them,
- or there exists an action $i \in B$ that is the first action along ρ to discard the timer started by the last action $i' \in B'$ and $\gamma' = \bullet$, *i.e.*, the timer of B' (re)started by i' reaches value zero when i discards it.

We also say that the actions i and i' participate in this race.

The first case of the race definition appears in Figure 9.2a (repeated in the margin), while the second case appears in Figure 9.2c (also repeated). In particular, B_1 and B_3 participate in a race. The nondeterminism is highlighted in Figures 9.2a and 9.2b (also repeated in the margin) where two actions (i and *to*$[x]$) occur at the same time but are processed in a different order in each figure. Unfortunately, imposing a particular way of resolving races (*i.e.*, imposing a particular action order) may seem arbitrary when modelling real-world systems. It is therefore desirable for the set of sequences of actions along timed runs to be independent to the resolution of races.







Definition 9.6.2 (Race-avoiding)**.** An MMT $\mathcal{M}$ is *race-avoiding* if and only if for all padded timed runs $\rho \in$ *ptruns*$(\mathcal{M})$ with races, there exists some $\rho' \in$ *ptruns*$(\mathcal{M})$ with no races such that *untime*(ρ') = *untime*(ρ).

Example 9.6.3. Let us come back to the timed run ρ of Example 9.2.5 that contains a race (see Figure 9.2a, which is the first figure repeated in the margin):

$$\begin{aligned}\rho = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 0, x_2 = 2) \xrightarrow{0} (q_2, x_1 = 0, x_2 = 2)\\ &\xrightarrow[\bot]{to[x_1]/o'} (q_0, \emptyset) \xrightarrow{1.5} (q_0, \emptyset).\end{aligned}$$

By moving the second occurrence of action i slightly earlier in ρ (we say that we *wiggle* the corresponding block), we obtain the timed run ρ':

$$\begin{aligned}\rho' = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i} (q_1, x_1 = 1) \xrightarrow{0.7} (q_1, x_1 = 0.3)\\ &\xrightarrow[(x_2,2)]{i} (q_2, x_1 = 0.3, x_2 = 2) \xrightarrow{0.3} (q_2, x_1 = 0, x_2 = 1.7)\\ &\xrightarrow[\bot]{to[x_1]} (q_0, \emptyset) \xrightarrow{1.5} (q_0, \emptyset).\end{aligned}$$

Notice that $\mathit{untime}(\rho') = q_0 \xrightarrow{i} q_1 \xrightarrow{i} q_2 \xrightarrow{to[x_1]} q_0 = \mathit{untime}(\rho)$. Moreover, ρ' contains no races as indicated in Figure 9.7.

Notice that several blocks could participate in the same race. The notion of block has been defined for padded timed runs only, as we do not want to consider runs that end *abruptly* during a race (some pending timeouts may not be processed at the end of the timed run, for instance). Moreover, it is always possible for the first delay to be positive as no timer is active in the initial state. Finally, non-zero delays at the start and the end of the runs allow to wiggle the blocks.

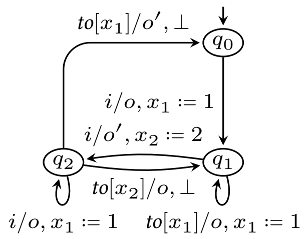
Let us now give the theorem that we prove throughout this section. In short, the PSPACE-hardness comes from a reduction from the acceptance problem of LBTMs (akin to what was done for the reachability problem), while the 3EXP membership is obtained via a monadic second-order formula.

Theorem 9.6.4. *Deciding whether an MMT is race-avoiding is* PSPACE*-hard and in* 3EXP*. It is in* PSPACE *if the sets of actions I and of timers X are fixed.*

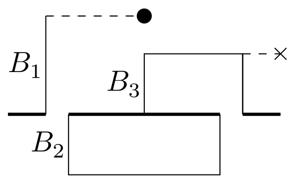
The formal proof is given in Section C.7 but requires tools that we now introduce. In Section 9.6.1, we formally define the idea of wiggling blocks, while Section 9.6.2 studies a characterization of when an MMT has a run in which zero delays are required. Section 9.6.3 gives an MSO formula that encodes our decision problem and which is used to show the 3EXP upper bound. Finally, Section 9.6.4 lists a few sufficient conditions that are easier to test.

9.6.1. Wiggling a timed run

First, let us formalize how to wiggle a padded timed run ρ of $\mathcal{M}$. Our approach is to study how to slightly move blocks along the time line of ρ in a way to get another $\rho' \in \mathit{ptruns}(\mathcal{M})$ where the races are eliminated while keeping the actions in the same order as in ρ. We call this action *wiggling*. We give an example before formalizing this notion.



Example 9.6.5. We consider again the MMT of Figure 9.1, which is repeated in the margin. We have seen in Example 9.6.3 and Figure 9.7 that the block B_2 of ρ can be slightly moved to the left to obtain the timed run ρ' with no race such that $untime(\rho) = untime(\rho')$. Figure 9.7 illustrates how to move B_2 by changing some of the delays.
In contrast, this is not possible for the timed run π of Example 9.2.7. Indeed looking at Figure 9.2c (also repeated in the margin), we see that it is impossible to move block B_2 to the left due to its race with B_1 (remember that we need to keep the same action order). It is also not possible to move it to the right due to its race with B_3. Similarly, it is impossible to move B_1 neither to the right (due to its race with B_2), nor to the left (otherwise its timer will time out instead of being discarded by B_3). Finally, one can also check that block B_3 cannot be moved.



Given a padded timed run ρ reading $d_1 \cdot i_1 \cdots i_n \cdot d_{n+1}$ and a block $B = (k_1 \dots k_m, \gamma)$ of ρ participating in a race, we say that we can *wiggle* if for some ϵ, we can move B to the left (by $\epsilon < 0$) or to the right (by $\epsilon > 0$), and obtain a timed run $\rho' \in ptruns(\mathcal{M})$ such that

- $untime(\rho) = untime(\rho') = i_1 \cdots i_n$, and
- B no longer participates in any race.

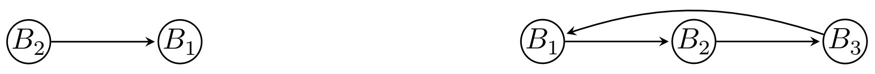
Definition 9.6.6 (Wiggle). Given a padded timed run ρ reading $d_1 \cdot i_1 \cdots i_n \cdot d_{n+1}$ and a block $B = (k_1 \dots k_m, \gamma)$ of ρ participating in a race, we say that we can *wiggle* if there are some $\epsilon \in \mathbb{R}$ and padded timed run ρ' reading $d'_1 \cdot i_1 \cdots i_n \cdot d'_{n+1}$ such that

- for all $i_{k_\ell} \in B$ with $k_\ell > 1$, if $i_{k_\ell - 1} \notin B$ (the action before i_{k_ℓ} in ρ does not belong to B), then $d'_{k_\ell} = d_{k_\ell} + \epsilon$,
- if there exists $i_{k_\ell} \in B$ with $k_\ell = 1$ (the first action of B is the first action of ρ), then $d'_1 = d_1 + \epsilon$,
- for all $i_{k_\ell} \in B$ with $k_\ell < n$, if $i_{k_\ell + 1} \notin B$ (the action after i_{k_ℓ} in ρ does not belong to B), then $d'_{k_\ell + 1} = d_{k_\ell + 1} - \epsilon$,
- if there exists $i_{k_\ell} \in B$ with $k_\ell = n$ (the last action of B is the last action of ρ), then $d'_{n+1} = d_{n+1} - \epsilon$,
- for all other d'_k, we have $d'_k = d_k$.

As $\rho' \in ptruns(\mathcal{M})$ and $untime(\rho) = untime(\rho')$, we must have $d'_k \geq 0$ for all k and $d'_1, d'_{n+1} > 0$.
We say that we can wiggle ρ, or that ρ is *wigglable*, if it is possible to wiggle its blocks, *one block at a time*, to obtain $\rho' \in ptruns(\mathcal{M})$ with no races such that $untime(\rho) = untime(\rho')$.

Observe that to wiggle B we move every action of B. Moreover, if all padded timed runs with races of an MMT $\mathcal{M}$ are wigglable, then $\mathcal{M}$ is race-avoiding.

In the next sections, we first associate a graph with any $\rho \in ptruns(\mathcal{M})$ in a way to characterize when ρ is wigglable thanks to this graph. We then state the equivalence between the race-avoiding characteristic of an MMT and the property that all $\rho \in ptruns(\mathcal{M})$ can be wiggled (Theorem 9.6.11). This



(a) Graph G_ρ. **(b)** Graph G_π.

Figure 9.8: Block graphs of the timed runs ρ and π of Example 9.2.7.

allows us to provide logic formulas to determine whether an MMT has an unwigglable run, and then to prove the upper bound of Theorem 9.6.4. We also discuss its lower bound. Finally, we discuss some sufficient hypotheses for a race-avoiding MMT.

9.6.2. Block graph and characterization

Given an MMT $\mathcal{M}$ and a padded timed run $\rho \in \mathit{ptruns}(\mathcal{M})$, we now study the conditions required to be able to wiggle ρ. For this purpose, we define the following graph G_ρ associated with ρ. When two blocks B and B' of ρ participate in a race, we write $B \prec B'$ if there exist actions $i \in B$ and $i' \in B'$ such that i, i' participate in this race and, according to Definition 9.6.1:

- either i occurs before i' along ρ and the total delay between i and i' is zero,
- or the timer of B' (re)started by i' reaches value zero when i discards it.

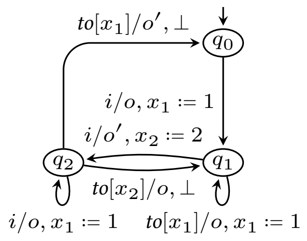
Definition 9.6.7 (Block graph)**.** The *block graph* of a padded timed run ρ is the tuple $G_\rho = (V, E)$ where

- V is the set of blocks of ρ, and
- $E \subseteq V \times V$ is such that there is an edge (B, B') if and only if $B \prec B'$.

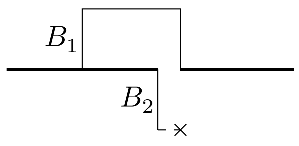
Example 9.6.8. Let $\mathcal{M}$ be the MMT from Figure 9.1, and ρ and π be the timed runs from Example 9.2.7:

$$\begin{aligned}
\rho = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0)\\
&\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 0, x_2 = 2) \xrightarrow{0} (q_2, x_1 = 0, x_2 = 2)\\
&\xrightarrow[\bot]{\mathit{to}[x_1]/o'} (q_0, \emptyset) \xrightarrow{1.5} (q_0, \emptyset).\\
\pi = (q_0, \emptyset) &\xrightarrow{0.5} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1)\\
&\xrightarrow[(x_2,2)]{i/o'} (q_2, x_1 = 1, x_2 = 2) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1)\\
&\xrightarrow[(x_1,1)]{i/o} (q_2, x_1 = 1, x_2 = 1) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 0)\\
&\xrightarrow[\bot]{\mathit{to}[x_2]/o} (q_1, x_1 = 0) \xrightarrow{0} (q_1, x_1 = 0)\\
&\xrightarrow[(x_1,1)]{\mathit{to}[x_1]/o} (q_1, x_1 = 1) \xrightarrow{0.5} (q_1, x_1 = 0.5).
\end{aligned}$$

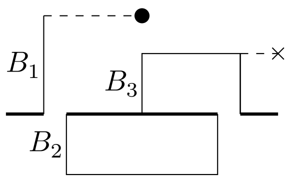
The block decompositions of π and ρ are represented in Figures 9.2a and 9.2c



$\mathit{to}[x_1]/o', \bot$
q_0
$i/o, x_1 := 1$
$i/o', x_2 := 2$
q_2
q_1
$\mathit{to}[x_2]/o, \bot$
$i/o, x_1 := 1$
$\mathit{to}[x_1]/o, x_1 := 1$



and are repeated in the margin. For the run ρ, it holds that $B_2 \prec B_1$ leading to the block graph G_ρ depicted in Figure 9.8a. For the run π, we get the block graph G_π depicted in Figure 9.8b.
Notice that G_ρ is acyclic while G_π is cyclic. By the following proposition, this difference is enough to characterize that ρ can be wiggled and π cannot.



Proposition 9.6.9. *Let $\mathcal{M}$ be an MMT and $\rho \in ptruns(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.*

Intuitively, a block cannot be moved to the left (resp. right) if it has a predecessor (resp. successor) in the block graph, due to the races in which it participates. Hence, if a block has both a predecessor and a successor, it cannot be wiggled (see Figures 9.2c and 9.8b for instance). Then, the blocks appearing in a cycle of the block graph cannot be wiggled. The other direction holds by observing that we can do a topological sort of the blocks if the graph is acyclic. We then wiggle the blocks, one by one, according to that sort. A complete proof is provided in Section C.4.

The next corollary is illustrated by Figure 9.9 with the simple cycle $(B_0, B_1, B_2, B_3, B_4, B_0)$.

Corollary 9.6.10. *Let $\mathcal{M}$ be an MMT and $\rho \in ptruns(\mathcal{M})$ be a padded timed run with races. Suppose that G_ρ is cyclic. Then there exists a cycle $\mathcal{C}$ in G_ρ such that*

- *any block of $\mathcal{C}$ participates in exactly two races described by this cycle,*
- *for any race described by $\mathcal{C}$, exactly two blocks of $\mathcal{C}$ participate in the race,*
- *the blocks $B = (k_1 \dots k_m, \gamma)$ of $\mathcal{C}$ satisfy either $m \geq 2$, or $m = 1$ and $\gamma = \bullet$.*

Proof. As G_ρ is cyclic, we consider a cycle of minimal length.
First notice that a block B of this cycle can only appear once per race. Indeed, the value at which a timer is (re)started in the block is positive, thus imposing non-zero delays between two actions of B (*i.e.*, two actions of B can not participate in a common race).
Second, by minimality of its length, the cycle is simple, implying that each of its blocks participates in exactly two races, one with its unique successor (in the cycle) and another one with its unique predecessor.
Third, assume that three blocks B_1, B_2 and B_3 participate in a common race, in that order. By the previous remark, they are pairwise distinct, and it must be that $B_1 \prec B_2$, $B_2 \prec B_3$ and $B_1 \prec B_3$. It follows that we get a smaller cycle by eliminating B_2, which is a contradiction.
Finally, assume that the cycle contains some block $B = (k_1, \gamma)$ with $\gamma \neq \bullet$. Let $B_1 \prec B$ (resp. $B \prec B_2$) be the predecessor (resp. successor) of B in the cycle. Due to the form of B, the three blocks B_1, B_2 and B participate in the same race, which is impossible. □

Proposition 9.6.9. Let $\mathcal{M}$ be an MMT and $\rho \in ptruns(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.

From the definition of wiggling, we know that if all padded timed runs with races of an MMT $\mathcal{M}$ are wigglable, then $\mathcal{M}$ is race-avoiding. The converse also holds as stated in the next theorem. By Proposition 9.6.9, this means that an

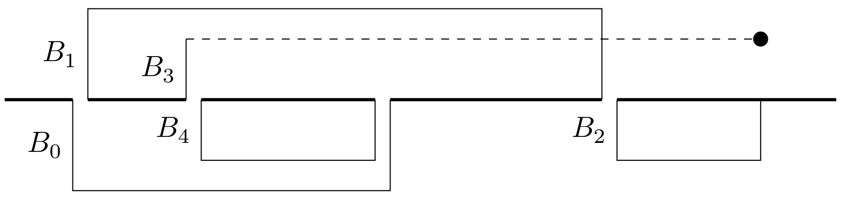


Figure 9.9: Races of a padded timed run ρ with $B_\ell \prec B_{\ell+1 \bmod 5}$, $0 \leq \ell \leq 4$.

MMT is race-avoiding if and only if the block graph of all its padded timed run is acyclic.

Theorem 9.6.11. *An MMT $\mathcal{M}$ is race-avoiding*

- *if and only if any padded timed run $\rho \in ptruns(\mathcal{M})$ with races can be wiggled,*
- *if and only if for any padded timed run $\rho \in ptruns(\mathcal{M})$, its block graph G_ρ is acyclic.*

Let us sketch the proof given in Section C.5. Clearly, the second equivalence follows from Proposition 9.6.9. Hence, it is sufficient to focus on the first equivalence. By modifying $\rho \in ptruns(\mathcal{M})$ to explicitly encode when a timer is discarded, one can show the races of ρ cannot be avoided if the block graph of ρ is cyclic as follows. Given two actions i, i' of this modified run, it is possible to define the *relative elapsed time* between i and i', denoted by $\text{reltime}(i, i')$, from the sum d of all delays between i and i': if i occurs before i', then $\text{reltime}(i, i') = d$, otherwise $\text{reltime}(i, i') = -d$. Lifting this to a sequence of actions from ρ is defined naturally. Then, one can observe that the relative elapsed time of a cyclic sequence of actions is zero, *i.e.*, $\text{reltime}(i_1, i_2, \dots, i_k, i_1) = 0$. Finally, from a cycle of G_ρ as described in Corollary 9.6.10, we extract a cyclic sequence of actions and prove, thanks to the concept of relative elapsed time, that any run ρ' such that $untime(\rho) = untime(\rho')$ must contain some races.

9.6.3. Existence of an unwigglable run

In this section, we give the intuition for the announced complexity bounds for the problem of deciding whether an MMT $\mathcal{M}$ is race-avoiding (Theorem 9.6.4).

Let us begin with the 3EXP-membership. The crux of our approach is to use the characterization of the race-avoiding property given in Theorem 9.6.11, to work with a slight modification of the region automaton $\mathcal{R}(\mathcal{M})$ of $\mathcal{M}$, and to construct a finite-state automaton whose language is the set of runs of $\mathcal{R}(\mathcal{M})$ whose block graph is cyclic. Hence, deciding whether $\mathcal{M}$ is race-avoiding amounts to deciding whether the language accepted by the latter automaton is empty. To do so, we construct a *monadic second-order* (MSO, for short; see [GTW03; Tho97] for an introduction) formula that is satisfied by words w labeling a run ρ of $\mathcal{R}(\mathcal{M})$ if and only if the block graph of ρ is cyclic.

[GTW03]: Grädel et al. (2003), *Automata, logics, and infinite games: a guide to current research*
[Tho97]: Thomas (1997), "Languages, Automata, and Logic"

Our *modification* of $\mathcal{R}(\mathcal{M})$ is best seen as additional annotations on the states and transitions of $\mathcal{R}(\mathcal{M})$. We extend the alphabet Σ as follows:

- we add a timer to each action $i \in \hat{\Sigma}$ to remember the updated timers

- we also use new symbols $\mathit{di}[x]$, $x \in X$, with the intent of explicitly encoding in $\mathcal{R}(\mathcal{M})$ when the timer x is discarded while its value is zero.

Therefore, the modified alphabet is

$$\Sigma = \{\tau\} \cup (\hat{\Sigma} \times (X \cup \{\bot\})) \cup \{\mathit{di}[x] \mid x \in X\}.$$

As a transition in $\mathcal{M}$ can discard more than one timer, we store the set D of discarded timers in the states of $\mathcal{R}(\mathcal{M})$, as well as outgoing transitions labeled by $\mathit{di}[x]$, for all discarded timers x. For this, the states of $\mathcal{R}(\mathcal{M})$ become

$$Q^{\mathcal{R}(\mathcal{M})} = \{(q, [\![\kappa]\!]_{\cong}, D) \mid q \in Q^{\mathcal{M}}, \kappa \in \mathsf{Val}(\chi(q)), D \subseteq X\}$$

and $\delta^{\mathcal{R}(\mathcal{M})}$ is modified in the natural way so that D is updated as required. Note that the size of this modified $\mathcal{R}(\mathcal{M})$ is only larger than what is stated in Lemma 9.5.4 by a factor of $2^{|X|}$.

Lemma 9.5.4. Let $\mathcal{M}$ be an MMT and $\mathcal{R}(\mathcal{M})$ be its region automaton. For a timer $x \in X$, c_x denotes the largest constant to which x is updated in $\mathcal{M}$. Let $C = \max_{x \in X} c_x$. Then, the number of states of $\mathcal{R}(\mathcal{M})$ is bounded by

$$|Q^{\mathcal{M}}| \cdot |X|! \cdot 2^{|X|} \cdot (C+1)^{|X|}.$$

Moreover, for all $q, q' \in Q^{\mathcal{M}}, \kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q)), \kappa' \in \mathsf{Val}(\chi^{\mathcal{M}}(q'))$, there exists a timed run from (q,κ) to (q',κ') in $\mathcal{M}$ if and only if there exists a run from $[\![(q,\kappa)]\!]_{\cong}$ to $[\![(q',\kappa')]\!]_{\cong}$ in $\mathcal{R}(\mathcal{M})$.

Note that any x-block $(i_{k_1}, \dots, i_{k_m}, \gamma)$ of a timed run ρ in $\mathcal{M}$ is translated into the sequence of symbols $(i'_{k_1}, \dots, i'_{k_m}, \gamma')$ in the corresponding run ρ' of the modified $\mathcal{R}(\mathcal{M})$ with an optional symbol γ' such that:

- $i'_{k_\ell} = (i_{k_\ell}, x)$, for $1 \leq \ell < m$,
- $i'_{k_m} = (i_{k_m}, \bot)$ if $\gamma = \bot$, and (i_{k_m}, x) otherwise,
- $\gamma' = \mathit{di}[x]$ if $\gamma = \bullet$, and γ' does not exist otherwise.

It follows that, instead of considering padded timed runs $\rho \in \mathit{ptruns}(\mathcal{M})$ and their block graph G_ρ, we work with their corresponding (padded) runs, blocks, and block graphs in the modified region automaton $\mathcal{R}(\mathcal{M})$ of $\mathcal{M}$.

Example 9.6.12. Let us consider the run π' of $\mathcal{R}(\mathcal{M})$ originally given in Example 9.5.5, *i.e.*,

$$\begin{aligned}\pi' = (q_0, [\![\emptyset]\!]_{\cong}) &\xrightarrow{\tau} (q_0, [\![\emptyset]\!]_{\cong}) \xrightarrow{i} (q_1, [\![x_1 = 1]\!]_{\cong}) \\ &\xrightarrow{i} (q_2, [\![x_1 = 1, x_2 = 2]\!]_{\cong}) \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 1]\!]_{\cong}) \\ &\xrightarrow{i} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}) \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 0]\!]_{\cong}) \\ &\xrightarrow{\mathit{to}[x_2]} (q_1, [\![x_1 = 0]\!]_{\cong}) \xrightarrow{\mathit{to}[x_1]} (q_1, [\![x_1 = 1]\!]_{\cong}) \\ &\xrightarrow{\tau} (q_1, [\![0 < x_1 < 1]\!]_{\cong}).\end{aligned}$$

With our modifications, it becomes

$$\begin{aligned}(q_0, [\![\emptyset]\!]_{\cong}, \emptyset) &\xrightarrow{\tau} (q_0, [\![\emptyset]\!]_{\cong}, \emptyset) \xrightarrow{(i,x_1)} (q_1, [\![x_1 = 1]\!]_{\cong}, \emptyset) \\ &\xrightarrow{(i,x_2)} (q_2, [\![x_1 = 1, x_2 = 2]\!]_{\cong}, \emptyset) \\ &\xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 1]\!]_{\cong}, \emptyset) \\ &\xrightarrow{(i,x_1)} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}, \{x_1\}) \\ &\xrightarrow{\mathit{di}[x_1]} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}, \emptyset) \\ &\xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 0]\!]_{\cong}, \emptyset)\end{aligned}$$

$$\xrightarrow{(\textit{to}[x_2],\bot)} (q_1, [\![x_1 = 0]\!]_{\cong}, \emptyset)$$
$$\xrightarrow{(\textit{to}[x_1],x_1)} (q_1, [\![x_1 = 1]\!]_{\cong}, \emptyset)$$
$$\xrightarrow{\tau} (q_1, [\![0 < x_1 < 1]\!]_{\cong}, \emptyset).$$

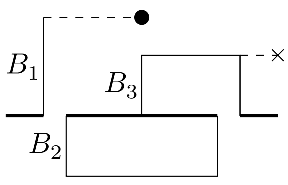
The transition with label *di*$[x_1]$ indicates that the timer x_1 is discarded in the original timed run while its value equals zero (see the race in which blocks B_1 and B_3 participate in Figure 9.2c, which is repeated in the margin). The three blocks of π become

$$B_1' = ((i, x_1), \textit{di}[x_1]),$$
$$B_2' = ((i, x_2), (\textit{to}[x_2], \bot)),$$

and

$$B_3' = ((i, x_1), (\textit{to}[x_1], x_1)).$$

The fact that in π, B_1 and B_2 participate in a race (with a zero-delay between their respective actions i and i), is translated in π' with the non existence of the τ-symbol between the symbols (i, x_1) and (i, x_2) in B_1' and B_2' respectively.



Proposition 9.6.13. *Let $\mathcal{M}$ be an MMT and $\mathcal{R}(\mathcal{M})$ be its modified region automaton. We can construct an MSO formula Φ of size linear in Σ and X such that a word labeling a run ρ of $\mathcal{R}(\mathcal{M})$ satisfies Φ if and only if ρ is a padded run that cannot be wiggled. Moreover, the formula Φ, in prenex normal form, has three quantifier alternations.*

The formula Φ of this proposition (proved in Section C.6) describes the existence of a cycle $\mathcal{C}$ of blocks $B_0, \dots, B_{k-1}$ such that $B_\ell \prec B_{\ell+1 \bmod k}$ for any $0 \leq \ell \leq k-1$, as described in Corollary 9.6.10 (see Figure 9.9). To do so, we consider the actions (*i.e.*, symbols of the alphabet Σ of $\mathcal{R}(\mathcal{M})$) participating in the races of $\mathcal{C}$: $i_0, i_1, \dots, i_{k-1}$ and $i_0', i_1', \dots, i_{k-1}'$ such that for all ℓ, i_ℓ, i_ℓ' belong to the same block, and $i_\ell', i_{\ell+1 \bmod k}$ participate in a race. One can write MSO formulas expressing that two actions participate in a race (there is no τ transition between them), that two actions belong to the same block, and, finally, the existence of these two action sequences.

From the formula Φ of Proposition 9.6.13, by the Büchi-Elgot-Trakhtenbrot theorem, we can construct a finite-state automaton whose language is the set of all words satisfying Φ. Its size is triple-exponential. We then compute the intersection $\mathcal{N}$ of this automaton with $\mathcal{R}(\mathcal{M})$ — itself exponential in size. Finally, the language of $\mathcal{N}$ is empty if and only if each padded timed run of $\mathcal{M}$ can be wiggled, and emptiness can be checked in polynomial time with respect to the triple-exponential size of $\mathcal{N}$, thus showing the 3EXP-membership of Theorem 9.6.4. Notice that when we fix the sets of inputs Σ and of timers X, the formula Φ becomes of constant size. Constructing $\mathcal{N}$ and checking its emptiness can be done "on the fly", yielding a nondeterministic decision procedure which requires a polynomial space only. We thus obtain that, under fixed inputs and timers, deciding whether an MMT is race-avoiding is in PSPACE.

Corollary 9.6.10. Let $\mathcal{M}$ be an MMT and $\rho \in \textit{ptruns}(\mathcal{M})$ be a padded timed run with races. Suppose that G_ρ is cyclic. Then there exists a cycle $\mathcal{C}$ in G_ρ such that

- any block of $\mathcal{C}$ participates in exactly two races described by this cycle,
- for any race described by $\mathcal{C}$, exactly two blocks of $\mathcal{C}$ participate in the race,
- the blocks $B = (k_1 \dots k_m, \gamma)$ of $\mathcal{C}$ satisfy either $m \geq 2$, or $m = 1$ and $\gamma = \bullet$.

Theorem 9.6.4. Deciding whether an MMT is race-avoiding is PSPACE-hard and in 3EXP. It is in PSPACE if the sets of actions I and of timers X are fixed.

The complexity lower bound of Theorem 9.6.4 follows from the PSPACE-hardness of the reachability problem for MMTs (see the intuition given in Section 9.5). We can show that any run in the MMT constructed from the given LBTM and word w can be wiggled. Once the designated state for the reachability reduction is reached, we add a widget that forces a run that cannot be wiggled. Therefore, as the only way of obtaining a run that cannot be wiggled is to reach a specific state (from the widget), the problem whether an MMT is race-avoiding is PSPACE-hard. Notice that this hardness proof is no longer valid if we fix the sets Σ and X.

A formal proof for Theorem 9.6.4 is given in Section C.7.

9.6.4. Sufficient hypotheses

Let us discuss some sufficient hypotheses for an MMT $\mathcal{M}$ to be race-avoiding.

1. If every state in $\mathcal{M}$ has at most one active timer, then $\mathcal{M}$ is race-avoiding. Up to renaming the timers, we actually have a single-timer MMT in this case.
2. If we modify the notion of timed run to impose non-zero delays everywhere in the run (even between two timeouts), then $\mathcal{M}$ is race-avoiding. Indeed, the only races that can appear are when a zero-valued timer is discarded, and it is impossible to form a cycle in the block graph with only this kind of races. Imposing a non-zero delay before a timeout is debateable. Nevertheless, imposing a non-zero delay before inputs only is not a sufficient hypothesis.
3. Let us fix a total order $<$ over the timers, and modify the semantics of an MMT to enforce that, in a race, any action of an x-block is processed before an action of a y-block, if $x < y$ (x is preemptive over y). Then, the MMT is race-avoiding. Towards a contradiction, assume there are blocks $B_0, B_1, \dots, B_{k-1}$ forming a cycle as described in Corollary 9.6.10, where each B_i is an x_i-block. By the order and the races, we get $x_0 \leq x_1 \leq \dots \leq x_{k-1} \leq x_0$, *i.e.*, we have a single timer (as in the first hypothesis). Hence, it is always possible to wiggle, which is a contradiction.

Corollary 9.6.10. Let $\mathcal{M}$ be an MMT and $\rho \in \mathit{ptruns}(\mathcal{M})$ be a padded timed run with races. Suppose that G_ρ is cyclic. Then there exists a cycle $\mathcal{C}$ in G_ρ such that

- any block of $\mathcal{C}$ participates in exactly two races described by this cycle,
- for any race described by $\mathcal{C}$, exactly two blocks of $\mathcal{C}$ participate in the race,
- the blocks $B = (k_1 \dots k_m, \gamma)$ of $\mathcal{C}$ satisfy either $m \geq 2$, or $m = 1$ and $\gamma = \bullet$.

9.7. Zones

In this section, we introduce *zones*, which are used to represent sets of valuations and are akin to the homonymous concept for timed automata (see Section 8.4). Then, for a given MMT $\mathcal{M}$, we construct its *zone MMT* which is itself an MMT. We then show multiple useful properties. Importantly, we obtain that $\mathcal{M}$ and its zone MMT are timed and symbolically equivalent. Furthermore, every run of a zone MMT is feasible.[15]

15: We recall that a run π is feasible if there exists a timed run ρ such that $\mathit{untime}(\rho) = \pi$.

Let X be a set of timers. A *zone Z over X* is a set of valuations over X, *i.e.*, $Z \subseteq \mathsf{Val}(X)$, described by the following grammar:

$$\phi ::= x < c \mid x \leq c \mid c < x \mid c \leq x \mid x - y < c \mid x - y \leq c \mid \phi_1 \wedge \phi_2$$

with $x, y \in X$ and $c \in \mathbb{N}$. It may be that a zone is empty. For the particular case $X = \emptyset$, we have that $\mathsf{Val}(X) = \{\emptyset\}$, meaning that a zone Z is either the zone $\{\emptyset\}$ or the empty zone.

Given a zone Z over X, a set $Y \subseteq X$, a timer x (that does not necessarily belong to X), and a constant $c \in \mathbb{N}^{>0}$, we define the following operations:

- The *downward closure* of Z where we let some time elapsed in all valuations of Z. That is, we obtain all valuations that can be reached from Z by just waiting (we consider delays that do not exceed the smallest value to avoid going below zero):

$$Z{\downarrow} = \{\kappa - d \mid \kappa \in Z, d \leq \min_{y \in \mathsf{dom}(\kappa)} \kappa(y)\}.$$

- The *restriction* of Z to $Y \neq \emptyset$ where, for every valuation of Z, we discard the values associated with timers in $X \setminus Y$, *i. e.*, we only keep the timers that are in Y:

$$Z\lceil Y = \{\kappa' \in \mathsf{Val}(Y) \mid \exists \kappa \in Z, \forall y \in Y : \kappa'(y) = \kappa(y)\}.$$

  If Y is empty, then we define the restriction as:

$$Z\lceil\emptyset = \begin{cases} \emptyset & \text{if } Z = \emptyset \\ \{\emptyset\} & \text{otherwise.} \end{cases}$$

- The *assignment* of x to c in the zone Z over X. Either x is already in X in which case we simply overwrite the value of x by c, or x is not in X in which case we "extend" the valuations of Z by adding x:

$$\begin{aligned} Z[x = c] = \{\kappa' \in \mathsf{Val}(X \cup \{x\}) \mid \kappa'(x) = c \wedge \\ \exists \kappa \in Z, \forall y \in X \setminus \{x\} : \kappa'(y) = \kappa(y)\}. \end{aligned}$$

- The *timeout of* x in Z where we keep all valuations of Z in which $x \in X$ times out:

$$\mathit{to}[Z, x] = \{\kappa \in Z \mid \kappa(x) = 0\}.$$

If Z is a zone, then $Z{\downarrow}$, $Z\lceil Y$, $Z[x = c]$, and $\mathit{to}[Z, x]$ are again zones [Cla+18]. Observe that $Z{\downarrow}$ and $\mathit{to}[Z, x]$ are zones over X (when $x \in X$), $Z\lceil Y$ is a zone over Y, and $Z[x = c]$ is a zone over $X \cup \{x\}$.

[Cla+18]: Clarke et al. (2018), *Handbook of Model Checking*

In the next example, we write $(x = c)$ to denote the zone over $X = \{x\}$ composed of the unique valuation κ such that $\kappa(x) = c$, and $c \leq x \leq c'$ as a shortcut for $c \leq x \wedge x \leq c'$ (when $c = c'$, we simply write $x = c$). Likewise, we write $c \leq x - y \leq c'$ instead of $c \leq x - y \wedge x - y \leq c'$. Moreover, negative constants are allowed, *i. e.*, $c \in \mathbb{Z}$. For instance, $-1 \leq x - y \leq 3$ is a shortcut for $(y - x \leq 1) \wedge (x - y \leq 3)$.

Example 9.7.1. The downward closure of the set $\{(x_1 = 1)\}$ is the zone

$$Z = \{(x_1 = 1)\}{\downarrow} = \{\kappa \mid 0 \leq \kappa(x_1) \leq 1\}.$$

This zone can be described by the constraint $0 \leq x_1 \leq 1$. Then, $\mathit{to}[Z, x_1] = \{(x_1 = 0)\}$. We can also restrict Z to either $\emptyset$ or $\{x_1\}$: $Z\lceil\emptyset = \{\emptyset\}$ and

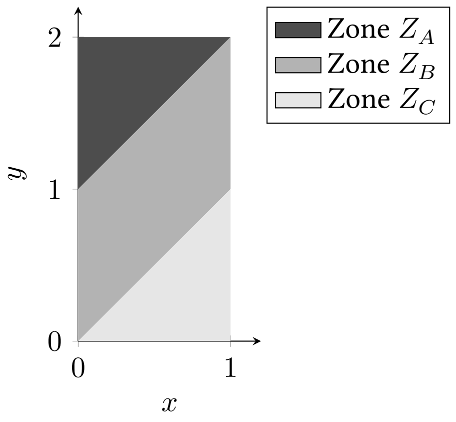


Figure 9.10: Visualization of some of the zones of Example 9.7.1.

$Z\lceil\{x_1\} = Z$.
Let us say we start a new timer x_2 with value 2:

$$\begin{aligned} Z[x_2 = 2] &= \{\kappa \mid 0 \leq \kappa(x_1) \leq 1 \wedge \kappa(x_2) = 2\} \\ &= (0 \leq x_1 \leq 1) \wedge (x_2 = 2). \end{aligned}$$

We then let time elapse to obtain the zone

$$\begin{aligned} Z_A &= (Z[x_2 = 2])\downarrow \\ &= ((0 \leq x_1 \leq 1) \wedge (x_2 = 2))\downarrow \\ &= (0 \leq x_1 \leq 1) \wedge (0 \leq x_2 \leq 2) \wedge (-2 \leq x_1 - x_2 \leq -1). \end{aligned}$$

This zone Z_A is visually represented as the dark gray part of Figure 9.10. While *to*$[Z_A, x_2]$ is empty (as x_1 is strictly smaller than x_2), we have that

$$\begin{aligned} \mathit{to}[Z_A, x_1] &= (x_1 = 0) \wedge (0 \leq x_2 \leq 2) \wedge (-2 \leq x_1 - x_2 \leq -1) \\ &= (x_1 = 0) \wedge (1 \leq x_2 \leq 2) \end{aligned}$$

is not empty. One could also consider these valuations where we restart x_1 to 1 (without necessarily letting it time out, *i.e.*, the valuation can be anywhere in the black area of Figure 9.10), and let time elapse:

$$\begin{aligned} Z_B &= (Z_A[x_1 = 1])\downarrow \\ &= ((x_1 = 1) \wedge (1 \leq x_2 \leq 2))\downarrow \\ &= (0 \leq x_1 \leq 1) \wedge (0 \leq x_2 \leq 2) \wedge (-1 \leq x_1 - x_2 \leq 0), \end{aligned}$$

which is represented in Figure 9.10 in light gray. This time, the set of valuations of Z_B where x_2 times out is not empty:

$$\begin{aligned} \mathit{to}[Z_B, x_2] &= (0 \leq x_1 \leq 1) \wedge (x_2 = 0) \wedge (-1 \leq x_1 - x_2 \leq 0) \\ &= (x_1 = 0) \wedge (x_2 = 0). \end{aligned}$$

Likewise, there are valuations of Z_B where x_1 times out:

$$\begin{aligned} \mathit{to}[Z_B, x_1] &= (x_1 = 0) \wedge (0 \leq x_2 \leq 2) \wedge (-1 \leq x_1 - x_2 \leq 0) \\ &= (x_1 = 0) \wedge (0 \leq x_2 \leq 1). \end{aligned}$$

Let us compute what happens when we consider these valuations where x_1 times out and we restart x_1 once more:

$$\begin{aligned} Z_C &= ((\mathit{to}[Z_B, x_1])[x_1 = 1])\downarrow \\ &= (((x_1 = 0) \wedge (0 \leq x_2 \leq 1))[x_1 = 1])\downarrow \\ &= ((x_1 = 1) \wedge (0 \leq x_2 \leq 1))\downarrow \\ &= (0 \leq x_1 \leq 1) \wedge (0 \leq x_2 \leq 2) \wedge (0 \leq x_1 - x_2 \leq 1). \end{aligned}$$

Figure 9.10 represents this zone with the lightest gray. The set of valuations where x_1 times out is empty, while there are valuations where x_2 is equal to zero.
Finally, we highlight that the valuations on the lines forming the frontiers between two zones in Figure 9.10 belong to both zones. That is, any configuration κ such that $0 \leq \kappa(x_1) \leq 1, 0 \leq \kappa(x_2) \leq 2$, and $\kappa(x_1) - \kappa(x_2) = -1$ (*i. e.*, the frontier between Z_A and Z_B) belong to both Z_A and Z_B. In other words, a valuation can belong to multiple zones.

9.7.1. Zone Mealy machine with timers

Given a sound and complete MMT $\mathcal{M}$, we explain how to construct its *zone MMT* that we denote $\mathit{zone}(\mathcal{M})$. The states of $\mathit{zone}(\mathcal{M})$ are pairs (q, Z) where q is a state of $\mathcal{M}$ and Z is a zone included in $\mathsf{Val}(\chi(q))$. The idea to construct $\mathit{zone}(\mathcal{M})$ is to start from the pair $(q_0^{\mathcal{M}}, \{\emptyset\})$ and explore every outgoing transition of $q_0^{\mathcal{M}}$. In general, we want to define the outgoing transitions of the current pair (q, Z). To do so, we consider the outgoing transitions of q in the complete machine $\mathcal{M}$.

- For every $q \xrightarrow[u]{i} q'$ with $i \in I$, we reproduce the same transition in $\mathit{zone}(\mathcal{M})$ (as it is always possible to trigger an input transition). That is, we define $(q, Z) \xrightarrow[u]{i} (q', Z')$ with a zone Z' that depends on the update: if $u = \bot$, then Z' is obtained by the restriction of Z to the active timers of q'; if $u = (x, c)$, then we first assign x to c. In both cases, we also let time elapse, *i. e.*, we always compute the downward closure.
- We perform the same idea with every $q \xrightarrow{\mathit{to}[x]}$ such that $x \in \chi(q)$, except that we first only consider the valuations of Z where x is zero. If $\mathit{to}[Z, x]$ is empty, we do not define the transition.[16]

16: Observe that x must be an enabled timer of q. However, there may be enabled timers of q that cannot reach zero in the zone Z, *i. e.*, it is a necessary but not sufficient condition.

Hence, in this way, we construct the states (q, Z) of $\mathit{zone}(\mathcal{M})$ such that $Z \neq \emptyset$ and that are reachable from its initial state $(q_0^{\mathcal{M}}, \{\emptyset\})$.

Definition 9.7.2 (Zone MMT)**.** Let $\mathcal{M}$ be a sound and complete MMT. We first define the tuple $\mathcal{Z} = (I^{\mathcal{Z}}, O^{\mathcal{Z}}, X^{\mathcal{Z}}, Q^{\mathcal{Z}}, q_0^{\mathcal{Z}}, \chi^{\mathcal{Z}}, \delta^{\mathcal{Z}})$ with:

- $I^{\mathcal{Z}} = I^{\mathcal{M}}$, $O^{\mathcal{Z}} = O^{\mathcal{M}}$, and $X^{\mathcal{Z}} = X^{\mathcal{M}}$,

- $Q^{\mathcal{Z}} = \{(q, Z) \mid q \in Q^{\mathcal{M}}, Z \subseteq \mathsf{Val}(\chi^{\mathcal{M}}(q)), Z \neq \emptyset\}$,
- $q_0^{\mathcal{Z}} = (q_0^{\mathcal{M}}, \{\emptyset\})$,
- For any $(q, Z) \in Q^{\mathcal{Z}}$, we define $\chi^{\mathcal{Z}}((q, Z)) = \chi^{\mathcal{M}}(q)$, *i. e.*, we simply copy the active timers of q,
- Let $(q, Z) \in Q^{\mathcal{Z}}$ and $q \xrightarrow[u]{i/o} q'$ be a transition of $\mathcal{M}$. We define

$$Z' = \begin{cases} Z & \text{if } u = \bot \text{ and } i \in I \\ Z[x = c] & \text{if } u = (x, c) \text{ and } i \in I \\ \mathit{to}[Z, x] & \text{if } u = \bot \text{ and } i = \mathit{to}[x] \\ (\mathit{to}[Z, x])[x = c] & \text{if } u = (x, c) \text{ and } i = \mathit{to}[x]. \end{cases}$$

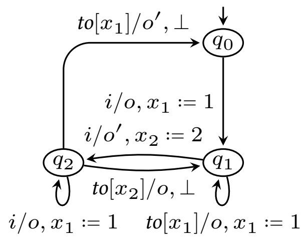
  Then, if $Z' \neq \emptyset$, we restrict Z' to $\chi^{\mathcal{M}}(q')$ and let time elapse. That is, we define

$$\delta^{\mathcal{Z}}((q, Z), i) = \left((q', (Z' \lceil \chi^{\mathcal{M}}(q'))\!\downarrow), o, u\right).$$

The *zone MMT of* $\mathcal{M}$, noted $\mathit{zone}(\mathcal{M})$, is then the MMT $\mathcal{Z}$ restricted to its reachable states.

Observe that the set of actions of $\mathit{zone}(\mathcal{M})$ is the set of actions of $\mathcal{M}$, *i. e.*, $A(\mathit{zone}(\mathcal{M})) = A(\mathcal{M})$.

Example 9.7.3. Let $\mathcal{M}$ be the MMT of Figure 9.1, which is repeated in the margin. Recall that $\chi^{\mathcal{M}}(q_0) = \emptyset$, $\chi^{\mathcal{M}}(q_1) = \{x_1\}$, and $\chi^{\mathcal{M}}(q_2) = \{x_1, x_2\}$. We construct its zone MMT $\mathit{zone}(\mathcal{M})$. We start with the state $q_0^{\mathcal{Z}} = (q_0, \{\emptyset\})$. Since there is no active timer, it is sufficient to define the i-transition:



$\mathit{to}[x_1]/o', \bot$
q_0
$i/o, x_1 := 1$
$i/o', x_2 := 2$
q_2
q_1
$\mathit{to}[x_2]/o, \bot$
$i/o, x_1 := 1$
$\mathit{to}[x_1]/o, x_1 := 1$

$$(q_0, \{\emptyset\}) \xrightarrow[(x_1,1)]{i/o} (q_1, 0 \leq x_1 \leq 1).$$

We can then define the $\mathit{to}[x_1]$- and the i-transition of the state $(q_1, 0 \leq x_1 \leq 1)$:

$$(q_1, 0 \leq x_1 \leq 1) \xrightarrow[(x_1,1)]{\mathit{to}[x_1]/o} (q_1, 0 \leq x_1 \leq 1)$$

and

$$\begin{aligned}(q_1, x_1 \leq 2) \xrightarrow[(x_2,2)]{i/o'} (q_2, (0 \leq x_1 \leq 1) \wedge (0 \leq x_2 \leq 2)& \\ \wedge (-2 \leq x_1 - x_2 \leq -1)).&\end{aligned}$$

Observe that the reached zone is the zone Z_A of Example 9.7.1. By the same example, we know that x_2 cannot time out in Z_A. Let us define the $\mathit{to}[x_1]$-transition by first computing the target zone, knowing that $q_2 \xrightarrow[\bot]{\mathit{to}[x_1]/o'} q_0$:

$$\begin{aligned}((\mathit{to}[Z_A, x_1])\lceil\{x_2\})\!\downarrow &= (((x_1 = 0) \wedge (1 \leq x_2 \leq 2))\lceil\{x_2\})\!\downarrow \\ &= 1 \leq x_2 \leq 2\!\downarrow \\ &= 0 \leq x_2 \leq 2.\end{aligned}$$

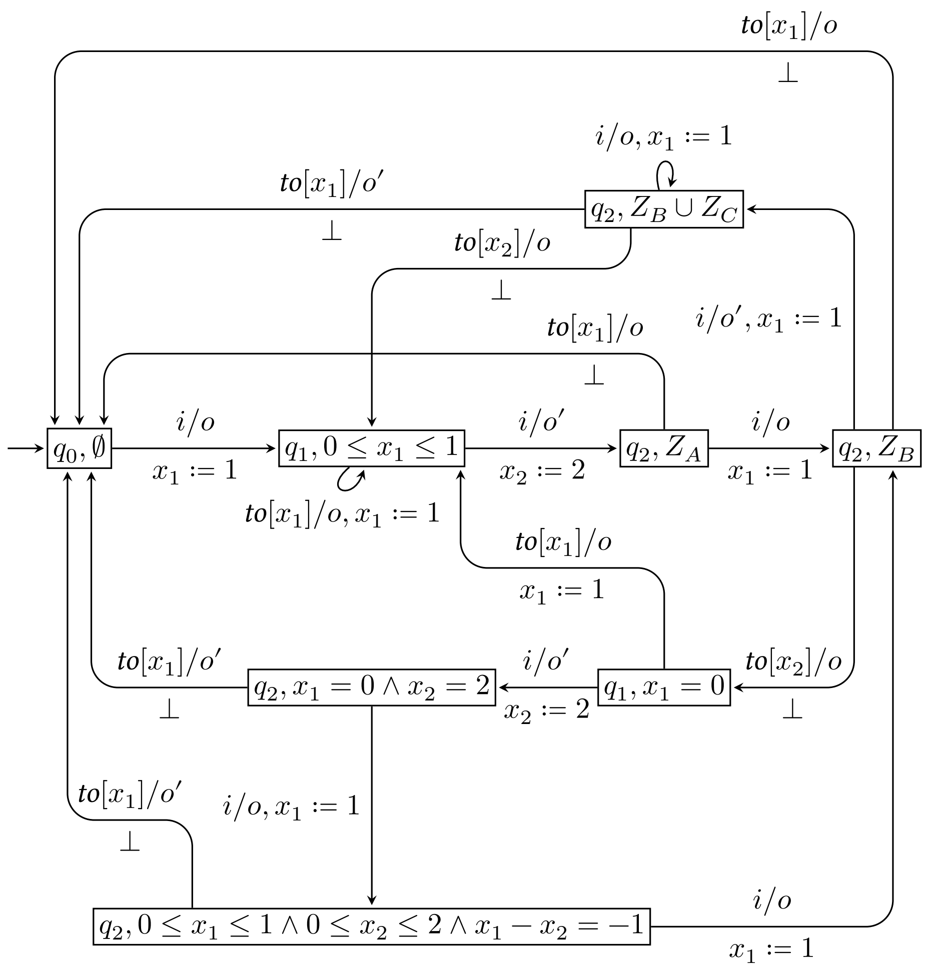


Figure 9.11: The zone MMT of the MMT of Figure 9.1.

Hence, we define the transition

$$(q_2, Z_A) \xrightarrow[\perp]{to[x_1]/o'} (q_0, \{\emptyset\}).$$

Let us now focus on defining the i-transition from (q_2, Z_A). As $q_2 \xrightarrow[(x_1,1)]{i/o} q_2$ in $\mathcal{M}$, the resulting zone is

$$Z_B = (0 \leq x_1 \leq 1) \wedge (0 \leq x_2 \leq 2) \wedge (-1 \leq x_1 - x_2 \leq 0),$$

by Example 9.7.1. So, we define

$$(q_2, Z_A) \xrightarrow[(x_1,1)]{i/o} (q_2, Z_B).$$

Figure 9.11 gives the zone MMT of $\mathcal{M}$.

Let us now argue that *zone*($\mathcal{M}$) has finitely many states and it is sound (when $\mathcal{M}$ is sound).

Lemma 9.7.4. *Let $\mathcal{M}$ be a sound MMT. Then, zone$(\mathcal{M})$ has finitely many states and is sound.*

Proof. The zone MMT is clearly sound because its transitions mimics the transitions of $\mathcal{M}$ and for any $(q, Z) \in Q^{\mathcal{Z}}$, we have $\chi^{\mathcal{Z}}((q, Z)) = \chi^{\mathcal{M}}(q)$. By construction, the states (q, Z) of *zone*$(\mathcal{M})$ are such that Z is a zone over $\chi^{\mathcal{M}}(q)$, described as a finite conjunction of constraints of the shape $x \bowtie c$ or $x - y \bowtie c$, with $\bowtie \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{N}$. For each timer x, let c_x be the maximal constant appearing on an update (re)starting x. Since the value of a timer can only decrease, it is clear that we will never reach a zone where $x > c_x$. Moreover, as the value of a timer must remain at least zero at any time, we also have a lower bound. In other words, we know that each timer x will always be confined between zero and c_x. From the shape of the constraints and these bounds, it follows immediately that there are finitely many zones. Hence, *zone*$(\mathcal{M})$ has finitely many states. □

The following theorem states multiple properties of the zone MMT. Its proof is given in Section C.8.

Theorem 9.7.5. *Let $\mathcal{M}$ be a complete and sound MMT, and zone$(\mathcal{M})$ be its zone MMT. Then,*

- *both MMTs $\mathcal{M}$ and zone$(\mathcal{M})$ have the same timed behaviors, i.e., it holds that for every timed word w, state $q \in Q^{\mathcal{M}}$, and valuation $\kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q))$,*
$$(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w} (q, \kappa) \in \mathit{truns}(\mathcal{M})$$
if and only if there exists a zone Z over $\chi^{\mathcal{M}}(q)$ such that $\kappa \in Z$ and
$$((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{w} ((q, Z), \kappa) \in \mathit{truns}(\mathit{zone}(\mathcal{M})).$$
- *$\mathcal{M}$ and zone$(\mathcal{M})$ have the same feasible runs,*
- *zone$(\mathcal{M})$ is sound and complete,*
- *$\mathcal{M}$ and zone$(\mathcal{M})$ are symbolically equivalent, and*
- *any run of zone$(\mathcal{M})$ is feasible.*

We then immediately obtain that $\mathcal{M}$ and *zone*$(\mathcal{M})$ are timed equivalent by Proposition 9.4.5.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

9.8. Conclusion

In this chapter, we presented Mealy machines with timers and their semantics. We showed that one can construct a timed Mealy machine (using clocks) from an MMT. Furthermore, we proved that deciding whether a state is reachable by some timed run remains PSPACE-complete, and adapted the notions of regions and zones to timers. Finally, we studied how to decide whether an MMT contains runs requiring non-zero delays to be observed, *i.e.*, whether an MMT is race-avoiding.

For future work, one may try to tighten the complexity bounds for the latter decision problem, both when fixing the sets I and X and when not. Furthermore, while we showed that a timed Mealy machine can be constructed from an MMT, we do not know about the other direction. That is, the sub-family of timed Mealy machines that is equivalent to MMTs is yet unknown. Finally, by Proposition 9.4.5, we know that symbolic equivalence implies timed equivalence, for any pair of MMTs. We showed in Section C.2 that the other direction does not hold, when the MMTs are *not* race-avoiding. It may be interesting to study whether both definitions are equivalent, when we can assume that all delays are positive.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

Active Learning of Mealy Machines with Timers 10.

In the previous chapter, we introduced *Mealy machines with timers* (MMTs) for which we now provide an active learning algorithm, based on [Bru+24]. The idea is to learn the timed system *symbolically*, *i. e.*, by abstracting away the delays. We show that this symbolic approach can be implemented via concrete timed runs, when the target MMT is race-avoiding.

[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

Our learning algorithm is an extension of the $L^{\#}$ algorithm for Mealy machines. We refer to Section 3.3 for an introduction to $L^{\#}$. That is, we use an observation tree (which is itself a partial MMT) to store the (symbolic) runs of the MMT of the teacher. Within this tree, we identify states that expose different (timed) behaviors in their subtrees (*i. e.*, the notion of *apartness* of $L^{\#}$), allowing us to construct a complete MMT hypothesis. Technical details and proofs are deferred to Appendix C.

Chapter contents

10.1. Introduction

Extending model learning algorithms to a setting that incorporates quantitative timing information turns out to be challenging. Twenty years ago, the first papers on this subject were published [GJL04; MP04], but we still do not have scalable algorithms for a general class of timed models. Consequently, in applications of model learning technology timing issues still need to be artificially suppressed.

[GJL04]: Grinchtein et al. (2004), "Learning of Event-Recording Automata"
[MP04]: Maler et al. (2004), "On Recognizable Timed Languages"

Several authors have proposed active learning algorithms for the popular framework of timed automata (TA; see Chapter 8) [AD94], which extends DFAs with clock variables. Some of these proposals, for instance [GJP06; GJL10; HJM20], have not been implemented due to their high complexity (both in terms of time and memory, and in the theoretical point of view). In recent years, however, several algorithms have been proposed and implemented that successfully learned realistic benchmark models. A first line of work restricts to subclasses of TAs such as deterministic one-clock timed automata (DOTAs) [An+20; XAZ22]. A second line of work explores synergies between active and passive learning algorithms. Aichernig et al. [Tap+19; APT20], for instance, employ a passive learning algorithm based on genetic programming to generate hypothesis models, which are subsequently refined using equivalence queries. A major result was obtained recently by Waga [Wag23], who presents an algorithm for active learning of (general) deterministic TAs and shows the effectiveness of the algorithm on various benchmarks. Waga's algorithm is inspired by ideas of Maler and Pnueli [MP04]. In particular, based on the notion of elementary languages of [MP04], Waga uses *symbolic queries*, which are then implemented using finitely many *concrete queries*. A challenge for learning algorithms for timed automata is the inference of the guards and resets that label transitions. As a result, the algorithm of Waga [Wag23] requires an exponential number of concrete membership queries to implement a single symbolic query.

[AD94]: Alur et al. (1994), "A Theory of Timed Automata"

[GJP06]: Grinchtein et al. (2006), "Inference of Event-Recording Automata Using Timed Decision Trees"

[GJL10]: Grinchtein et al. (2010), "Learning of event-recording automata"

[HJM20]: Henry et al. (2020), "Active Learning of Timed Automata with Unobservable Resets"

[An+20]: An et al. (2020), "Learning One-Clock Timed Automata"

[XAZ22]: Xu et al. (2022), "Active Learning of One-Clock Timed Automata Using Constraint Solving"

[Tap+19]: Tappler et al. (2019), "Time to Learn - Learning Timed Automata from Tests"

[APT20]: Aichernig et al. (2020), "From Passive to Active: Learning Timed Automata Efficiently"

[Wag23]: Waga (2023), "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization"

In the previous chapter, we introduced Mealy machines with timers (MMT), which form a subfamily of timed automata with outputs, and we claimed that MMTs are easier to learn. It is noteworthy that Jonsson and Vaandrager already tried to devise an active learning algorithm for MMTs based on L^* [JV18], although their work was never published. This chapter provides an active learning algorithm for MMTs based on $L^{\#}$ [Vaa+22] (see Section 3.3), and using ideas of Maler and Pnueli [MP04]. Experiments with a prototype implementation, written in Rust, show that our algorithm is able to efficiently learn realistic benchmarks.

[JV18]: Jonsson et al. (2018), *Learning mealy machines with timers*

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

This chapter is structured as follows. Section 10.2 describes the basic learning framework for MMTs, in particular the three types of symbolic queries that an MMT learner may use, and claim that these symbolic queries can be performed with finitely many concrete output and equivalence queries. Then, Section 10.3 describes the observation tree that our algorithm uses to record the outcomes of symbolic queries. It also lifts the notion of apartness from the $L^{\#}$ algorithm to the timed setting. Two states of an observation tree are apart if they can not correspond to the same state of the hidden MMT. The notion of apartness is parametrized by a matching, which specifies how the timers of both states correspond to each other. Section 10.4 describes our new learning algorithm for MMTs, called $L^{\#}_{\text{MMT}}$. Analogous to $L^{\#}$, the $L^{\#}_{\text{MMT}}$ algorithm maintains a growing set of *basis states*, states in the observation tree that are pairwise apart, for any possible matching. Basis states will act as states in a hypothesis MMT that will be constructed by the learner. Successors of basis states that are not in the basis themselves are called *frontier* states. Like in $L^{\#}$, we perform queries in order to establish apartness of frontier states from as many basis states as possible, for as many matchings as possible. If a frontier state is apart from all basis states, for all possible matchings, then $L^{\#}_{\text{MMT}}$ may extend the basis. If

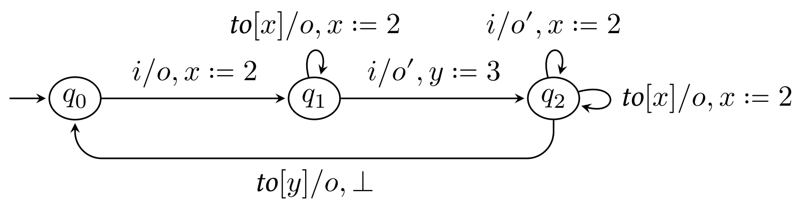


Figure 10.1: Running example MMT with $\chi(q_0) = \emptyset$, $\chi(q_1) = \{x\}$, $\chi(q_2) = \{x, y\}$.

there exists a basis state and a matching for which a frontier state is not apart, this provides $L^{\#}_{\text{MMT}}$ with information on how it may construct a hypothesis model. The mechanism for hypothesis construction for MMTs is closely related to the chronometric relational morphisms of Maler and Pnueli [MP04]. Finally, Section 10.5 discusses our prototype implementation and the results for the benchmarks. Technical details and proofs are deferred to Appendix C.

Running example and outline of $L^{\#}_{\text{MMT}}$

Throughout this chapter, we use the MMT $\mathcal{M}$ of Figure 10.1 as our running example. One can check that $\mathcal{M}$ is sound, complete, and race-avoiding. However, $\mathcal{M}$ has some unfeasible runs. For instance, $q_0 \xrightarrow{i \cdot i \cdot to[y]}$ is not feasible, due to the fact that y is started with a bigger value than x.

As we did for MMs (see Section 3.3.1), one can unfold the runs of $\mathcal{M}$ into an infinite tree (of finite arity), rooted at $q_0^{\mathcal{M}}$:

- consider all outgoing i-transitions of a state q with $i \in I$, create a new state p, and define the transition $q_0^{\mathcal{M}} \xrightarrow{i} p$;
- let π be the unique (due to the tree-shaped structure) run from the initial state to q, consider all $to[x]$-transitions such that x is enabled after π (*i.e.*, $x \in \chi_0(\pi)$), and again create a new state and transition; and
- repeat this for every new state p.

We claim that we can construct a finite MMT from this infinite tree.[1]

1: By identifying a finite subset of states, called the *basis*, as is the case for $L^{\#}$ (see Section 3.3.1).

The goal of the learner in our algorithm $L^{\#}_{\text{MMT}}$ is to infer a finite part of this infinite tree such that this part is sufficiently big to be able to construct an MMT $\mathcal{N}$ that is timed equivalent to $\mathcal{M}$ (which can be checked via an equivalence query). However, as a timer must be enabled after the run from the initial state for a timeout transition to be defined, the tree may lack some transitions compared to its corresponding state in $\mathcal{M}$. For instance, the run $q_0^{\mathcal{M}} \xrightarrow{i \cdot i \cdot to[y]}$ cannot be reproduced in the tree (as the value of y is necessarily bigger than the value of x). As we will argue in the next sections, this is problematic, as it means that some states of the tree have strictly less enabled timers than their corresponding states in the teacher's MMT.

Hence, we assume that the teacher's MMT is feasible.[2] In particular, this can be achieved by constructing its zone MMT, as introduced in Section 9.7. Let us recall the main theorem over the zone MMT:

2: Recall that an MMT $\mathcal{M}$ is feasible when every run of $\mathcal{M}$ is feasible.

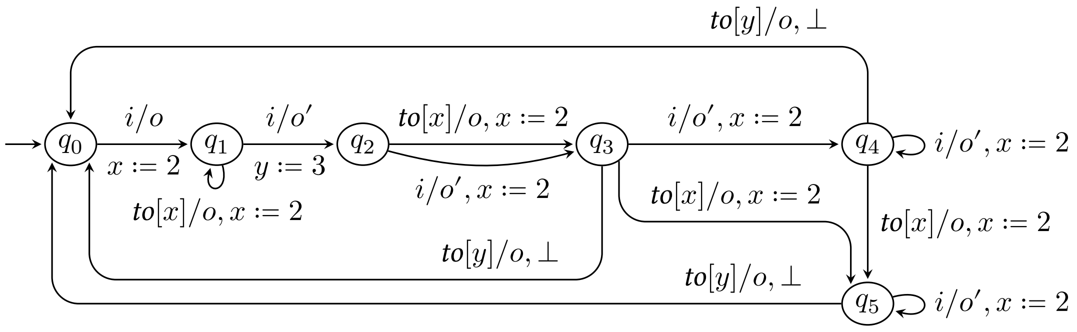


Figure 10.2: The zone MMT of the MMT of Figure 10.1, with $\chi(q_0) = \emptyset$, $\chi(q_1) = \{x\}$, and $\chi(q) = \{x, y\}$ for all the other states q.

Theorem 9.7.5. *Let $\mathcal{M}$ be a complete and sound MMT, and zone$(\mathcal{M})$ be its zone MMT. Then,*

- *both MMTs $\mathcal{M}$ and zone$(\mathcal{M})$ have the same timed behaviors, i.e., it holds that for every timed word w, state $q \in Q^{\mathcal{M}}$, and valuation $\kappa \in$ $\mathsf{Val}(\chi^{\mathcal{M}}(q))$,*
$$(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w} (q, \kappa) \in \mathit{truns}(\mathcal{M})$$
if and only if there exists a zone Z over $\chi^{\mathcal{M}}(q)$ such that $\kappa \in Z$ and
$$((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{w} ((q, Z), \kappa) \in \mathit{truns}(\mathit{zone}(\mathcal{M})).$$
- *$\mathcal{M}$ and zone$(\mathcal{M})$ have the same feasible runs,*
- *zone$(\mathcal{M})$ is sound and complete,*
- *$\mathcal{M}$ and zone$(\mathcal{M})$ are symbolically equivalent, and*
- *any run of zone$(\mathcal{M})$ is feasible.*

Figure 10.2 gives the zone MMT of $\mathcal{M}$.

We fix a set of inputs I and a set of outputs O and assume that both the learner and the teacher use these sets.

10.2. Learning framework

As usual for learning algorithms, we rely on Angluin's framework [Ang87]: we assume we have a teacher who knows an MMT $\mathcal{M}$, and a learner who does not know $\mathcal{M}$ but can query the teacher to obtain knowledge about $\mathcal{M}$. However, we need $\mathcal{M}$ to satisfy some constraints: it must be sound, complete, race-avoiding, and feasible. We say that $\mathcal{M}$ is *s-learnable*[3] when it satisfies the four requirements. We highlight that an s-learnable MMT can be constructed from an MMT that is already sound, complete, and race-avoiding, by computing its zone MMT. The following proposition is a direct consequence of Theorem 9.7.5.

[Ang87]: Angluin (1987), "Learning Regular Sets from Queries and Counterexamples"

3: The *s* stands for *symbolically*.

Proposition 10.2.1. *For any sound, complete, and race-avoiding MMT $\mathcal{M}$, there exists an s-learnable MMT $\mathcal{N}$ such that $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$.*

By Proposition 9.4.5, we then obtain that $\mathcal{M}$ and $\mathcal{N}$ are timed equivalent.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

In this section, we describe the queries a learner for MMTs can use. First of all, let us recall the queries for (classical) Mealy machines [Vaa+22; SG09]:

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"
[SG09]: Shahbaz et al. (2009), "Inferring Mealy Machines"

Definition 3.3.1. Let $\mathcal{M}$ be the MM of the teacher. A learner for MMs can use two types of *queries*:

- An *output query*, denoted by $\mathbf{OQ}(w)$, with w a word over I, returns $output^{\mathcal{M}}(w)$.
- An *equivalence query*, denoted by $\mathbf{EQ}(\mathcal{H})$, with $\mathcal{H}$ an MM, returns
  - **yes** if $\mathcal{M} \approx \mathcal{H}$, and
  - a word w over I such that $output^{\mathcal{M}}(w) \neq output^{\mathcal{H}}(w)$ otherwise. Such a word w is called a *counterexample*.

We adapt those queries to take into account the timed behavior induced by the timers of $\mathcal{M}$. We do it in two ways: one that uses timed input words (see Section 9.4.1), and one that uses symbolic words (see Section 9.4.2). Finally, we claim that symbolic queries can be performed with finitely many concrete queries, when $\mathcal{M}$ is race-avoiding (which is why that aspect is required in the definition of s-learnable).

10.2.1. Concrete queries

The concrete queries are a direct adaptation of the queries for classical MM: one requests the output of a timed word, and one asks whether a hypothesis is correct.

Definition 10.2.2 (Concrete queries for MMTs)**.** Let $\mathcal{M}$ be the sound, complete, and race-avoiding MMT of the teacher. A learner for MMTs can use two types of *concrete queries*:

- $\mathbf{OQ}(w)$ with w a tiw such that $toutputs^{\mathcal{M}}(w) \neq \emptyset$: the teacher outputs a tow in $toutputs^{\mathcal{M}}(w)$.
- $\mathbf{EQ}(\mathcal{H})$ with $\mathcal{H}$ a sound and complete MMT $\mathcal{H}$: the teacher replies **yes** if $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{H}$, or a tiw w such that $toutputs^{\mathcal{M}}(w) \neq toutputs^{\mathcal{H}}(w)$.

As $\mathcal{M}$ is race-avoiding, we can assume without loss of generality that the returned counterexample w has a unique run in $\mathcal{M}$, *i.e.*, that $|toutputs^{\mathcal{M}}(w)| = 1$.[4] Figure 10.3 gives a visual representation of the adapted Angluin's framework for MMTs.

4: If it is not, it is possible to wiggle the blocks. See Section 9.6.

10.2.2. Symbolic queries

On the opposite, symbolic queries necessitate a symbolic word. While output and equivalence queries are easily adapted to the symbolic case (again, see Section 9.4.2), we cannot obtain information about enabled timers solely from them. Hence, we need a new type of query, called a *wait query*, in order to deal with timed behavior.

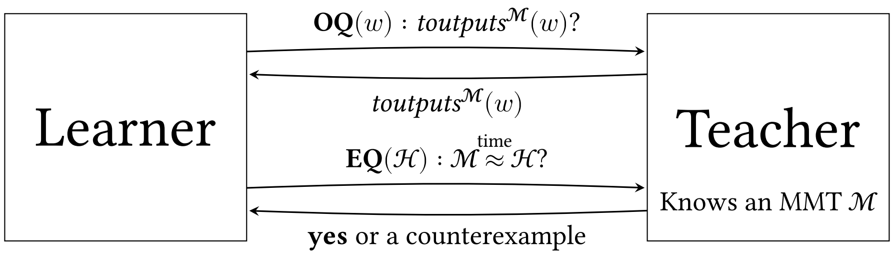

$\mathbf{OQ}(w)$: $\mathit{toutputs}^{\mathcal{M}}(w)$?
Learner
Teacher
$\mathit{toutputs}^{\mathcal{M}}(w)$
$\mathbf{EQ}(\mathcal{H})$: $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{H}$?
Knows an MMT $\mathcal{M}$
yes or a counterexample

Figure 10.3: Adaptation of Angluin's framework for MMTs, with concrete queries.

Definition 10.2.3 (Symbolic queries for MMTs)**.** Let $\mathcal{M}$ be the sound, complete, and race-avoiding MMT of the teacher. A learner for MMTs can use three types of *symbolic queries*

- A *symbolic output query*, denoted by $\mathbf{OQ}^{\mathrm{s}}(\mathtt{w})$, with $\mathtt{w}$ an sw such that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \in \mathit{runs}(\mathcal{M})$, returns the outputs of $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}}$.
- A *symbolic wait query*, denoted by $\mathbf{WQ}^{\mathrm{s}}(\mathtt{w})$, with $\mathtt{w}$ an sw such that
$$q_0^{\mathcal{M}} \xrightarrow{i_1} \cdots \xrightarrow{i_n} q_n \in \mathit{runs}(\mathcal{M})$$
with $\overline{i_1 \cdots i_n} = \mathtt{w}$, returns the set of all pairs (j, c) such that $q_{j-1} \xrightarrow{i_j \cdots i_n \cdot \mathit{to}[x]}$ is x-spanning.
- A *symbolic equivalence query*, denoted by $\mathbf{EQ}^{\mathrm{s}}(\mathcal{H})$, with $\mathcal{H}$ a sound and complete MMT, returns **yes** if $\mathcal{H} \overset{\text{sym}}{\approx} \mathcal{M}$, or an sw $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ such that
  - either $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}} \in \mathit{runs}(\mathcal{H}) \Leftrightarrow q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \notin \mathit{runs}(\mathcal{M})$,
  - or there exists j such that
$$q_0^{\mathcal{M}} \xrightarrow{\mathtt{i}_1 \cdots \mathtt{i}_{j-1}} q \xrightarrow[u]{\mathtt{i}_j / o} \in \mathit{runs}(\mathcal{M}),$$
$$q_0^{\mathcal{H}} \xrightarrow{\mathtt{i}_1 \cdots \mathtt{i}_{j-1}} q' \xrightarrow[u']{\mathtt{i}_j / o'} \in \mathit{runs}(\mathcal{H}),$$
  and
    - $o \neq o'$ or
    - $u = (x, c)$, $u' = (x', c')$, $c \neq c'$, and $q \xrightarrow{\mathtt{i}_j \cdots \mathtt{i}_k}$ is x-spanning for some $k \in \{j+1, \ldots, n\}$.

$\mathbf{OQ}^{\mathrm{s}}$ and $\mathbf{EQ}^{\mathrm{s}}$ are analogous to regular output and equivalence queries in the setting of Mealy machines, while $\mathbf{WQ}^{\mathrm{s}}$ provides, for each timer x enabled at the end of the run induced by the given symbolic word, the transition which last (re)started x and the constant with which x was (re)started. Figure 10.4 gives a visual representation of the symbolic queries for MMTs.

To conclude this section, we claim that these three symbolic queries can be performed via concrete output and equivalence queries, *i.e.*, queries using tiws instead of sws, when $\mathcal{M}$ is sound, complete, and race-avoiding. A proof is provided in Section C.9. Note, however, that it requires properties on the data structure used during the learning process and introduced in the next section.

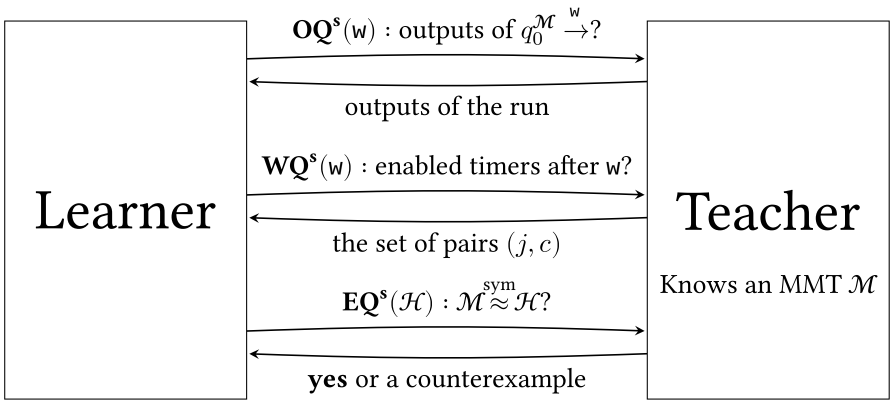

OQ^s(w) : outputs of $q_0^{\mathcal{M}} \xrightarrow{w}$?
outputs of the run
WQ^s(w) : enabled timers after w?
the set of pairs (j, c)
EQ$^s(\mathcal{H}) : \mathcal{M} \overset{\text{sym}}{\approx} \mathcal{H}$?
yes or a counterexample
Learner
Teacher
Knows an MMT $\mathcal{M}$

Figure 10.4: Adaptation of Angluin's framework for MMTs, with symbolic queries.

Proposition 10.2.4. *For any s-learnable MMTs, the three symbolic queries can be implemented via a polynomial number of concrete output and equivalence queries.*

10.3. Observation tree

In this section, we describe the main data structure of our learning algorithm, which is a modification of the *observation tree* used by $L^{\#}$ [Vaa+22] (see Section 3.3). Such a tree, denoted by $\mathcal{T}$, is a partial MMT that stores the observations obtained via symbolic queries. We impose that $\mathcal{T}$ is tree-shaped[5] and feasible. Each state q of $\mathcal{T}$ has its own timer x_q that can only be started by the incoming transition of q, and may only be restarted by a *to*$[x_q]$-transition. Thanks to its tree-shape nature, we can impose strict constraints on the set of active and enabled timers of a state q: a timer x is active in q if and only if there is an x-spanning run traversing q, and is enabled if and only if the *to*$[x]$-transition is defined from q.[6]

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

5: That is, every state has a unique incoming transition, except for the initial state. Hence, for every state q there is a unique run from the initial state to q.

6: See Section 9.2.3 for a way to compute the set of enabled timers, in general.

Definition 10.3.1 (Observation tree). An *observation tree* is a tree-shaped sound MMT $\mathcal{T} = (I, O, X, Q, q_0, \chi, \delta)$ such that

- $X = \{x_q \mid q \in Q \setminus \{q_0\}\}$,
- $\forall p \xrightarrow[(x,c)]{i} q$ with $i \in I : x = x_q$,
- every run of $\mathcal{T}$ is feasible,
- $\forall q \in Q, x \in X : x \in \chi(q)$ if and only if there is an x-spanning run traversing q, and
- $\forall q \in Q, x \in X : x \in \chi_0(q)$ if and only if $q \xrightarrow{to[x]} \in \mathit{runs}(\mathcal{T})$.

Example 10.3.2. Figure 10.5 gives an example of an observation tree $\mathcal{T}$. To ease the reading, we write x_i instead of x_{t_i} for all states t_i. The active timers are as follows: $\chi(t_1) = \chi(t_2) = \{x_1\}$, $\chi(t_3) = \chi(t_5) = \{x_1, x_3\}$, $\chi(t_6) = \{x_3, x_6\}$, and $\chi(t) = \emptyset$ for the other states t. In every case, we satisfy that $x \in \chi(t)$ if and only if there is an x-spanning run going through

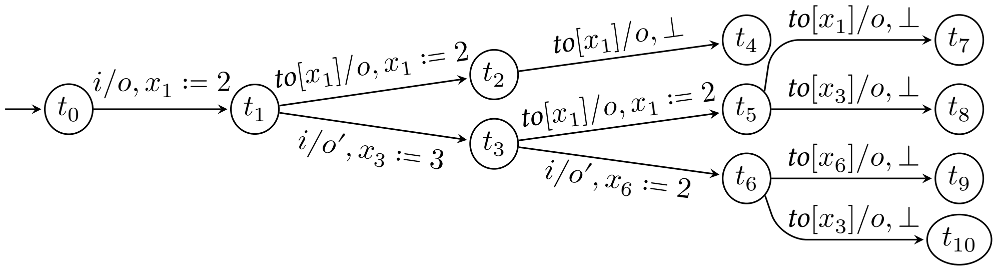


Figure 10.5: Sample observation tree (we write x_i instead of x_{t_i} for all states t_i) with $\chi(t_1) = \chi(t_2) = \{x_1\}$, $\chi(t_3) = \chi(t_5) = \{x_1, x_3\}$, $\chi(t_6) = \{x_3, x_6\}$, and $\chi(t) = \emptyset$ for the other states t.

t, and a transition $t \xrightarrow{to[x]}$ is defined if and only if $x \in \chi_0(t)$. Moreover, every run of $\mathcal{T}$ is feasible.

During the learning process, it is not always be possible to decide whether the update of a new transition must be (x, c) or $\bot$. Instead, we assume by default that a transition does nothing, *i.e.*, its update is $\bot$. Later on, we may discover that this transition actually (re)starts a timer, and its update will be replaced by some (x, c). Hence, in the rest of this section, we will use $\bot$ as a sort of wildcard. We first characterize the fact that the transitions of $\mathcal{T}$ must correspond to the transitions of some hidden MMT $\mathcal{M}$ (which, in Section 10.4, will be the MMT we want to learn), except that a $\bot$ in $\mathcal{T}$ may be a (x, c) in $\mathcal{M}$. The fact that not all updates are known implies that some active and enabled timers may be missing in $\mathcal{T}$ compared to $\mathcal{M}$. We then adapt, in Section 10.3.2, the notion of *apartness* of [Vaa+22] for MMTs.

10.3.1. Functional simulation

As we did for classical MMs with Definition 3.3.4 in order to link the (finitely many) runs of $\mathcal{T}$ to runs of $\mathcal{M}$, we introduce a function $f : Q^{\mathcal{T}} \to Q^{\mathcal{M}}$ that maps states of $\mathcal{T}$ to states of $\mathcal{M}$ such that every transition $q \xrightarrow{i}$ of $\mathcal{T}$ can be reproduced from $f(q)$ while producing the same output. In addition, since $\mathcal{T}$ and $\mathcal{M}$ may use different timers, we need a function $g : X^{\mathcal{T}} \rightharpoonup X^{\mathcal{M}}$ that maps every *active* timer of $\mathcal{T}$ to a timer of $\mathcal{M}$. We require that

- when a timer x is active in q, the corresponding timer $g(x)$ is active in $f(q)$,[7] and
- when x and y are distinct timers that are both active in some state q, we do not allow g to map x and y to the same timer of $\mathcal{M}$.

These conditions imply that

- the number of timers that are active in $f(q)$ is at least as large as the number of timers active in q,
- for any transition $q \xrightarrow[u]{i/o} q'$ in $\mathcal{T}$, there exists a transition $f(q) \xrightarrow[u']{i'/o} f(q')$ in $\mathcal{M}$ such that $i' = i$ if $i \in I$, or $i' = to[g(x)]$ if $i = to[x]$, *i.e.*, we read a corresponding action, output the same symbol and reach the state corresponding to q', and

Definition 3.3.4. Let $\mathcal{M}$ be a complete MM and $\mathcal{T}$ be an observation tree. A *functional simulation* $f : \mathcal{T} \to \mathcal{M}$ is a map $f : Q^{\mathcal{T}} \to Q^{\mathcal{M}}$ with

- $f(q_0^{\mathcal{T}}) = q_0^{\mathcal{M}}$, and
- for all $q, q' \in Q^{\mathcal{T}}$, $i \in I$, and $o \in O$, $q \xrightarrow{i/o} q'$ implies $f(q) \xrightarrow{i/o} f(q')$.

We say that $\mathcal{T}$ is an *observation tree for* $\mathcal{M}$ if there exists a functional simulation $f : \mathcal{T} \to \mathcal{M}$.

7: However, there may be active timers in $f(q)$ that are unknown in q.

- if $u = (x, c)$, then $u' = (g(x), c)$, *i.e.*, we do the same update. However, if $u = \bot$, we may not have found the actual update yet, so we do not impose anything on u' (it can be any update in $U(\mathcal{M})$).

Then, for a run π of $\mathcal{T}$, we can consider its corresponding run in $\mathcal{M}$ via f and g, denoted by $\langle f, g\rangle(\pi)$, which must preserve the spanning sub-runs.

Definition 10.3.3 (Functional simulation)**.** Let $\mathcal{T}$ be an observation tree and $\mathcal{M}$ be an s-learnable MMT. A *functional simulation* $\langle f, g\rangle : \mathcal{T} \to \mathcal{M}$ is a pair of a map

$$f : Q^{\mathcal{T}} \rightharpoonup Q^{\mathcal{M}}$$

and a map

$$g : \bigcup_{q \in Q^{\mathcal{T}}} \chi^{\mathcal{T}}(q) \to X^{\mathcal{M}}.$$

Let g be lifted to actions such that $g(i) = i$ for every $i \in I$, and $g(to[x]) = to[g(x)]$ for every $x \in \mathrm{dom}(g)$. We require that $\langle f, g\rangle$ preserves initial states, active timers, and transitions:

$$f(q_0^{\mathcal{T}}) = q_0^{\mathcal{M}} \tag{FS0}$$

$$\forall q \in Q^{\mathcal{T}}, \forall x \in \chi^{\mathcal{T}}(q) : g(x) \in \chi^{\mathcal{M}}(f(q)) \tag{FS1}$$

$$\forall q \in Q^{\mathcal{T}}, \forall x, y \in \chi^{\mathcal{T}}(q) : x \neq y \;\Rightarrow\; g(x) \neq g(y) \tag{FS2}$$

$$\forall q \xrightarrow[(x,c)]{i/o} q' : f(q) \xrightarrow[(g(x),c)]{g(i)/o} f(q') \tag{FS3}$$

$$\forall q \xrightarrow[\bot]{i/o} q' : f(q) \xrightarrow{g(i)/o} f(q'). \tag{FS4}$$

Thanks to (FS3) and (FS4), we lift $\langle f, g\rangle$ to runs in a straightforward manner. We require that for all runs π of $\mathcal{T}$,

$$\langle f, g\rangle(\pi) \text{ is } y\text{-spanning} \Rightarrow \exists x : \pi \text{ is } x\text{-spanning and } g(x) = y. \tag{FS5}$$

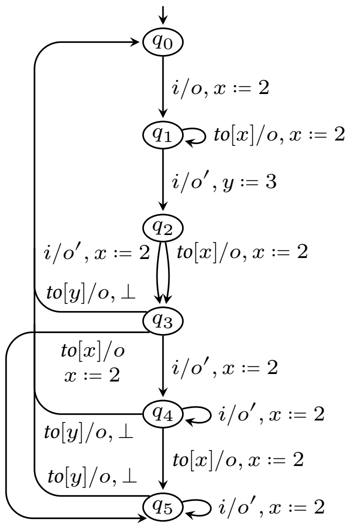
If there exists $\langle f, g\rangle : \mathcal{T} \to \mathcal{M}$, we say that $\mathcal{T}$ is an *observation tree for* $\mathcal{M}$.

Example 10.3.4. Let $\mathcal{M}$ be the MMT of Figure 10.2, which is repeated in the margin. Recall that $\mathcal{M}$ is s-learnable. Then, the observation tree $\mathcal{T}$ of Figure 10.5 (which is repeated on the next page) is an observation tree for $\mathcal{M}$ with the functional simulation $\langle f, g\rangle$ such that

$$\begin{aligned} f(t_0) = f(t_8) = f(t_{10}) &= q_0 \\ f(t_1) = f(t_2) = f(t_4) &= q_1 \\ f(t_3) &= q_2 \\ f(t_5) = f(t_6) &= q_3 \\ f(t_7) = f(t_9) &= q_5, \end{aligned}$$

and

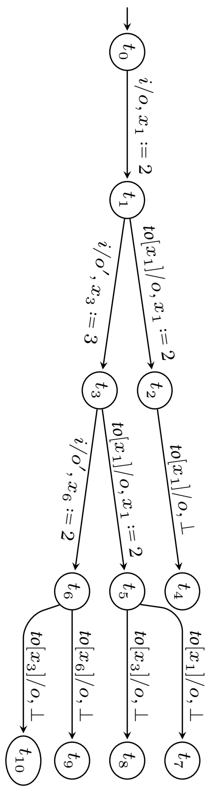
$$\begin{aligned} g(x_1) = g(x_6) &= x \\ g(x_3) &= y. \end{aligned}$$



Let $\pi = t_1 \xrightarrow{i \cdot to[x_1] \cdot to[x_3]}$, which is x_3-spanning. Observe that

$$\langle f, g \rangle(\pi) = q_1 \xrightarrow{i \cdot to[x] \cdot to[y]}$$

is y-spanning and π is x_3-spanning. As $g(x_3) = y$, (FS5) is satisfied.



Observe that for fixed $\mathcal{T}$ and $\mathcal{M}$ there exists at most one functional simulation. Further properties can be deduced from the definition of $\langle f, g \rangle$.

Corollary 10.3.5. *Let $\mathcal{T}$ be an observation tree for an s-learnable $\mathcal{M}$ with $\langle f, g \rangle$. Then, for all states $q \in Q^{\mathcal{T}}$, we have:*

- $|\chi^{\mathcal{T}}(q)| \leq |\chi^{\mathcal{M}}(f(q))|$, *and*
- *for all* $x \in \chi_0^{\mathcal{T}}(q)$, $g(x) \in \chi_0^{\mathcal{M}}(f(q))$.

Proof. We start with the first part, *i.e.*, $|\chi^{\mathcal{T}}(q)| \leq |\chi^{\mathcal{M}}(f(q))|$. By (FS1), we have that any timer x that is active in q is such that $g(x)$ is active in $f(x)$. Moreover, by (FS2), $g(x) \neq g(y)$ for any $x \neq y \in \chi^{\mathcal{T}}(q)$. So, it is not possible for q to have more active timers than $f(q)$.
Now, the second part, *i.e.*, $\forall x \in \chi_0^{\mathcal{T}}(q) : g(x) \in \chi_0^{\mathcal{M}}(f(q))$. Let $x \in \chi_0^{\mathcal{T}}(q)$. By definition of $\mathcal{T}$, it follows that $q \xrightarrow{to[x]}$ is defined. So, by (FS3) and (FS4), we have $f(q) \xrightarrow{to[g(x)]}$, meaning that $g(x) \in \chi_0^{\mathcal{M}}(f(q))$, as $\mathcal{M}$ is complete. □

Let us now characterize when a state q is deemed *explored*, in the sense that we know exactly its set of enabled timers, that is, when compared to $\mathcal{M}$. As we know that $\chi^{\mathcal{N}}(q_0^{\mathcal{N}}) = \emptyset$ for any sound MMT $\mathcal{N}$, it follows that $q_0^{\mathcal{T}}$ is always explored.

Definition 10.3.6 (Explored states). Let $q \in Q^{\mathcal{T}}$ and π be the unique run from $q_0^{\mathcal{T}}$ to q in $\mathcal{T}$. We say that q is *explored* if

$$|\chi_0^{\mathcal{T}}(q)| = |\chi_0^{\mathcal{M}}(f(q))|.$$

Define $\mathcal{E}^{\mathcal{T}}$ as the maximal set of explored states of $\mathcal{T}$ that induces a subtree containing $q_0^{\mathcal{T}}$, *i.e.*, $p \in \mathcal{E}^{\mathcal{T}}$ for all $p \xrightarrow{i} q$ with $q \in \mathcal{E}^{\mathcal{T}}$.

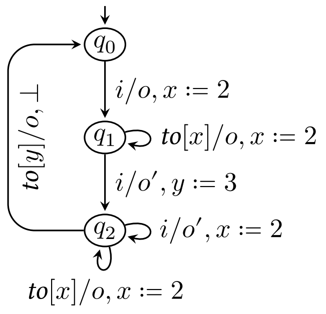
Observe that there exists a one-to-one correspondence between $\chi_0^{\mathcal{T}}(q)$ and $\chi_0^{\mathcal{M}}(f(q))$ when q is explored, by (FS2) and Corollary 10.3.5.

$$\forall q \in Q^{\mathcal{T}}, x, y \in \chi^{\mathcal{T}}(q) : \quad x \neq y \Rightarrow g(x) \neq g(y) \qquad \text{(FS2)}$$

A learning algorithm to construct $\mathcal{T}$ and to extend $\mathcal{E}^{\mathcal{T}}$, when needed, from a hidden MMT is given in Section 10.4. We now give an example for explored states, before arguing why we require the teacher's MMT to be feasible.

Example 10.3.7. Let $\mathcal{T}$ be the observation tree of Figure 10.5 (which is repeated in the margin above) and $\langle f, g \rangle$ the functional simulation of Example 10.3.4. We can define the set of explored states to be $\mathcal{E}^{\mathcal{T}} = \{t_0, t_1, t_2, t_3, t_5, t_6\}$. Notice that t_4 does not have any enabled timer, while $f(t_4) = q_1$ has an enabled timer. Thus, t_4 is indeed not explored.

Let us now explain why we require the teacher's MMT to be feasible. Let $\mathcal{T}$ be the observation tree of Figure 10.5 and $\mathcal{N}$ be the not-s-learnable MMT of Figure 10.1, which is repeated in the margin. We can still define the maps $f : Q^{\mathcal{T}} \to Q^{\mathcal{N}}$ and $g : X^{\mathcal{T}} \to X^{\mathcal{N}}$:



$$\begin{aligned} f(t_0) = f(t_8) = f(t_{10}) &= q_0 \\ f(t_1) = f(t_2) = f(t_4) &= q_1 \\ f(t_3) = f(t_5) = f(t_6) = f(t_7) = f(t_9) &= q_2, \end{aligned}$$

and

$$\begin{aligned} g(x_1) = g(x_6) &= x \\ g(x_3) &= y. \end{aligned}$$

We have $|\chi_0^{\mathcal{T}}(t_3)| = 1$ but $|\chi_0^{\mathcal{N}}(f(t_3))| = |\chi_0^{\mathcal{N}}(q_2)| = 2$. However, as every run of $\mathcal{T}$ must be feasible and x_3 cannot time out in t_3, it is impossible to get the equality. Therefore, in order to define explored states, we must require that $\mathcal{M}$ is s-learnable. Moreover, this definition of explored states is needed to have the one-to-one correspondence between $\chi_0^{\mathcal{T}}(q)$ and $\chi_0^{\mathcal{M}}(f(q))$, which is vital for some proofs. Finally, the notion of apartness we are about to define solely considers what happens *after* a state, *i.e.*, the transitions leading to the states are not considered at all. This means that, when we take the corresponding runs in the teacher's MMT, we know nothing about the active and enabled timers. That is, our (natural) approach for apartness is another argument towards our definition of explored states within the tree, and the requirements we impose to have an s-learnable MMT.

10.3.2. Apartness

The learning process, presented in Section 10.4, will construct an observation tree by using symbolic queries. In order to be able to construct a hypothesis from the tree, we need to distinguish the states that have clearly different timed behaviors, as is the case for $L^\#$ (see Section 3.3).

Let $\mathcal{T}$ be an observation tree for an s-learnable $\mathcal{M}$. Similar to the $L^\#$ algorithm for Mealy machines [Vaa+22], we define a notion of *apartness* for MMTs. In the setting of Mealy machines, states p, p' are apart (denoted $p \# p'$) if they have different output responses to the same input word. As our observation tree has timers, we need to handle the fact that different timers can represent the same timer in $\mathcal{M}$.[8] There are some easy cases, *e.g.*, if $p \xrightarrow{i/o}$ and $p' \xrightarrow{i/o'}$ such that $o \neq o'$ for some input i, then p and p' can be deemed apart. However, in general, we have to decide whether two timers $x \in \chi(p)$ and $y \in \chi(p')$ should be assumed equivalent. While, in theory, one could use the functional simulation $\langle f, g \rangle : \mathcal{T} \to \mathcal{M}$ to group together (or, *match*) timers x, y such that $g(x) = g(y)$, the functional simulation is unknown during learning. Hence, we need to guess how to group the timers. In order to do so, we consider *matchings* between the active timers of two states. We will also need to lift the notion of matching to encompass the timers started along a run, allowing us to finally define the apartness of states.

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

8: In our example, we have $g(x_1) = g(x_6)$, for instance.

Apartness of timers

First, recall that (FS2) states that if two distinct timers x, y are both active in the same state, $g(x) \neq g(y)$. That is, x and y must correspond to different timers in $\mathcal{M}$. Hence, we say that x, y are *apart*, denoted $x \mathrel{\#^t} y$,[9] whenever there exists $q \in Q^{\mathcal{T}}$ such that $x, y \in \chi^{\mathcal{T}}(q)$.

$$\forall q \in Q^{\mathcal{T}}, x, y \in \chi^{\mathcal{T}}(q) : \quad x \neq y \Rightarrow g(x) \neq g(y) \qquad \text{(FS2)}$$

9: The small t in the notation aims to distinguish between apartness of timers $\#^t$ and apartness of states $\#$, as introduced below.

Matchings

As said above, we need to guess which timers in different parts of $\mathcal{T}$ have to be considered "as one", *i.e.*, we *conjecture* that $g(x) = g(y)$. We rely on the concept of matching to encode this conjectured *equivalence* of timers. We first do it between the timers active in a pair of states, and then lift it to runs starting from these two states, in order to accommodate the fresh timers.

Given two finite sets S and T, a relation $m \subseteq S \times T$ is a *matching* from S to T if it is an injective partial function. We write $m : S \leftrightarrow T$ if m is a matching from S to T. A matching m is *maximal* if it is total or surjective.

Given two states p and p' of an observation tree $\mathcal{T}$, we consider a matching $m : \chi^{\mathcal{T}}(p) \leftrightarrow \chi^{\mathcal{T}}(p')$, denoted by abuse of notation as $m : p \leftrightarrow p'$. We say that m is *valid* if it is consistent with (FS2). More formally,

Definition 10.3.8 (Valid matching)**.** A matching $m : p \leftrightarrow p'$ is said to be *valid* if

$$\forall x \in \mathsf{dom}(m) : \neg(x \mathrel{\#^t} m(x)).$$

We lift m to actions:

$$m(i) = \begin{cases} i & \text{if } i \in I \\ \mathit{to}[m(x)] & \text{if } i = \mathit{to}[x] \text{ with } x \in \mathsf{dom}(m). \end{cases}$$

Let $\pi = p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} p_n$ and $\pi' = p'_0 \xrightarrow{i'_1} p'_1 \xrightarrow{i'_2} \cdots \xrightarrow{i'_n} p'_n$. We lift a matching $m : p_0 \leftrightarrow p'_0$ to runs π, π' starting from p_0 and p'_0 as follows. For π' to match π, we require that for all $j \in \{1, \ldots, n\}$:

- If $i_j \in I$, then $i'_j = i_j$.
- If $i_j = \mathit{to}[x]$ for some $x \in X^{\mathcal{T}}$ then there are two possibilities:
  - $x \in \chi^{\mathcal{T}}(p_0)$, in which case i'_j is $\mathit{to}[m(x)]$, or
  - $x = x_{p_k}$ for some k (*i.e.*, x is started along the run), in which case i'_j is $\mathit{to}[x_{p'_k}]$ with the same k.

  That is, i'_j must use the "same" timer according to m or the updates of the run.

When π and π' match, we write $m^{\pi}_{\pi'} : \pi \leftrightarrow \pi'$ with

$$m^{\pi}_{\pi'} = m \cup \{(x_{p_k}, x_{p'_k}) \mid k \leq n\}$$

and $i'_j = m^{\pi}_{\pi'}(i_j)$ for every j.[10] For a fixed $\pi \in \mathit{runs}(\mathcal{T})$ and m, there is at most one run $\pi' \in \mathit{runs}(\mathcal{T})$ such that $m^{\pi}_{\pi'} : \pi \leftrightarrow \pi'$. We denote by $\mathit{read}^{m}_{\pi}(p'_0)$ this

10: In the following, we will need to describe situations where x_{p_k} is started and not $x_{p'_k}$, or vice-versa. Hence, we define matchings over every timer, not just those that are actually started.

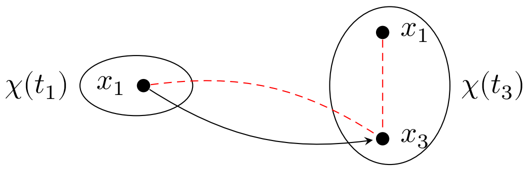


Figure 10.6: Visualization of the matching m of Example 10.3.9. The dashed lines represent the apartness of timers, *i.e.*, $x_1 \overset{t}{\#} x_3$.

unique run π' if it exists (if it does not, the function is left undefined). That is, the function "reads" π from p_0', using m to rename the timers.

Example 10.3.9. In the examples, we write $x \mapsto x', y \mapsto y'$ for the matching m such that $m(x) = x'$ and $m(y) = y'$.
Let $\mathcal{T}$ be the observation tree of Figure 10.5 (which is repeated in the margin) and

$$\pi = t_0 \xrightarrow[(x_1,2)]{i} t_1 \xrightarrow{to[x_1]} t_2 \in \mathit{runs}(\mathcal{T}).$$

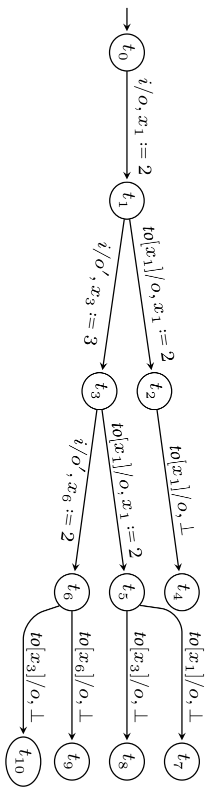
We compute $\mathit{read}_\pi^\emptyset(t_3)$ (where $\emptyset$ denotes the empty matching). The first symbol in π is i, *i.e.*, we take the transition $t_3 \xrightarrow[(x_6,2)]{i} t_6$. The second symbol in π is $to[x_1]$. Since x_1 was a fresh timer started along π, we retrieve the corresponding fresh timer in the new run, which is x_6. Hence,

$$\mathit{read}_\pi^\emptyset(t_3) = t_3 \xrightarrow{i} t_6 \xrightarrow{to[x_6]} t_9.$$

Let us now consider the run

$$\pi' = t_1 \xrightarrow{i} t_3 \xrightarrow{to[x_1]} t_5 \in \mathit{runs}(\mathcal{T}).$$

Let $m : t_1 \leftrightarrow t_3$ be the (maximal) matching such that $x_1 \mapsto x_3$. This matching is represented in Figure 10.6 where we use solid lines to represent the matching m and dashed lines for the apartness of timers. Observe that m is invalid, as $x_1, x_3 \in \chi(t_3)$. Nevertheless, $\mathit{read}_{\pi'}^m(t_3) = t_3 \xrightarrow{i} t_6 \xrightarrow{to[m(x_1)]=to[x_3]} t_{10}$.
Finally, let $m' : t_1 \leftrightarrow t_3$ such that $m' = x_1 \mapsto x_1$. As $m'(x_1)$ is not enabled in t_6, $\mathit{read}_{\pi'}^{m'}(t_3)$ is undefined.



Apartness of states

Two states p_0 and p_0' are apart under a matching m, if we have runs $\pi = p_0 \xrightarrow{w}$ and $\pi' = \mathit{read}_\pi^m(p_0')$ such that $m_{\pi'}^\pi$ is invalid (in which case we say that the apartness is *structural*), or the runs exhibit different behaviors (the apartness is *behavioral*).

Definition 10.3.10 (Apartness). Two states p_0, p_0' are m-*apart* with $m : p_0 \leftrightarrow p_0'$, denoted by $p_0 \#^m p_0'$, if there are $\pi = p_0 \xrightarrow{i_1} \cdots \xrightarrow[u]{i_n/o} p_n$ and

$\pi' = p'_0 \xrightarrow{i'_1} \cdots \xrightarrow[u']{i'_n/o'} p'_n$, with $m^{\pi}_{\pi'} : \pi \leftrightarrow \pi'$, and

Structural apartness there exists $x \in \mathsf{dom}(m^{\pi}_{\pi'})$ such that $x \not\# m^{\pi}_{\pi'}(x)$, or
Behavioral apartness one of the following holds:

$$
\begin{aligned}
&o \neq o' && \text{(outputs)}\\
&u = (x, c) \wedge u' = (x', c') \wedge c \neq c' && \text{(constants)}\\
&p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge |\chi_0(p_n)| \neq |\chi_0(p'_n)| && \text{(sizes)}\\
&p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge \exists x \in \mathsf{dom}(m^{\pi}_{\pi'}) : (x \in \chi_0(p_n) \\
&\qquad \Leftrightarrow m^{\pi}_{\pi'}(x) \notin \chi_0(p'_n)) && \text{(enabled)}
\end{aligned}
$$

The word $w = i_1 \ldots i_n \in A(\mathcal{T})^*$ is called a *witness* of $p_0 \#^m p'_0$, denoted by $w \vdash p_0 \#^m p'_0$.

Example 10.3.11. Let $\mathcal{T}$ be the observation tree of Figure 10.5, which is repeated in the margin. By Example 10.3.9, we have the following runs:

$$\pi = t_0 \xrightarrow[(x_1,2)]{i/o} t_1 \xrightarrow[(x_1,2)]{to[x_1]/o} t_2$$

and

$$read^{\emptyset}_{\pi}(t_3) = t_3 \xrightarrow[(x_6,2)]{i/o'} t_6 \xrightarrow[\perp]{to[x_6]/o} t_9.$$

As the first transition of π outputs o but the first transition of $read^{\emptyset}_{\pi}(t_3)$ outputs $o' \neq o$, we conclude that $i \vdash t_0 \#^{\emptyset} t_3$ by (outputs). Since $t_1, t_6 \in \mathcal{E}^{\mathcal{T}}$ (see Example 10.3.7) and

$$|\chi_0(t_1)| = 1 \neq |\chi_0(t_6)| = 2,$$

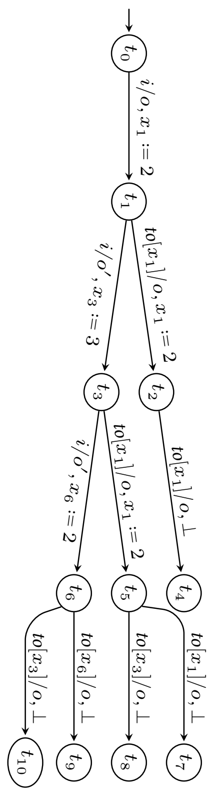
we have $\varepsilon \vdash t_1 \#^{\emptyset} t_6$ and $i \vdash t_0 \#^{\emptyset} t_3$ by (sizes).
Moreover,

$$\pi' = t_1 \xrightarrow[(x_3,3)]{i/o'} t_3 \xrightarrow[(x_1,2)]{to[x_1]/o} t_5$$

and

$$read^{x_1 \mapsto x_3}_{\pi'}(t_3) = t_3 \xrightarrow[(x_6,2)]{i/o'} t_6 \xrightarrow[\perp]{to[x_3]/o} t_{10}.$$

Since $x_1 \mapsto x_3$ is invalid (as $x_1 \not\# x_3$), $t_1 \#^{x_1 \mapsto x_3} t_3$ is structural. We also have $i \vdash t_1 \#^{x_1 \mapsto x_3} t_3$ due to (constants).
Recall that $read^{x_1 \mapsto x_1}_{\pi'}(t_3)$ is undefined as $x_1 \notin \chi_0(t_6)$ (see Example 10.3.9). As $t_3, t_6 \in \mathcal{E}^{\mathcal{T}}$, we thus have $i \vdash t_1 \#^{x_1 \mapsto x_1} t_3$ by (enabled).



Extensibility

Observe that if w is a witness of $p \#^m p'$ and $read^{m}_{p \xrightarrow{w}}(p')$ gives the run $p' \xrightarrow{w'}$, it holds that w' is a witness of $p' \#^{m^{-1}} p$. That is, while the definition of apartness of states is not symmetric, one can easily obtain it, by changing the witness.

Moreover, any extension m' of m is such that $w \vdash p \#^{m'} p'$, *i.e.*, taking a larger matching does not break the apartness, as

$$\mathit{read}^{m}_{p \xrightarrow{w}}(p') = \mathit{read}^{m'}_{p \xrightarrow{w}}(p') = p' \xrightarrow{w'} .$$

Lemma 10.3.12. *Let $m : p \leftrightarrow p'$ and $m' : p \leftrightarrow p'$ be two matchings such that $w \vdash p \#^{m} p'$ and $m \subseteq m'$. Then, $w \vdash p \#^{m'} p'$.*

Proof. Let $w \vdash p \#^{m} p'$ and $m \subseteq m'$. Moreover, let $p_0 = p$, $p'_0 = p'$, and $\pi = p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow[u]{i_n/o} p_n$ and $\pi' = \mathit{read}^{m}_{p \xrightarrow{w}}(p') = p'_0 \xrightarrow{i'_1} p'_1 \xrightarrow{i'_2} \cdots \xrightarrow[u']{i'_n/o'} p'_n$, with $m^{\pi}_{\pi'} : \pi \leftrightarrow \pi'$. By definition, each i_j is either an input, or $to[x]$ with $x \in \mathrm{dom}(m^{\pi}_{\pi'})$. Thus, since $m \subseteq m'$, it follows that $\mathit{read}^{m'}_{p \xrightarrow{w}}(p')$ uses exactly the same actions and takes the same transitions as $\mathit{read}^{m}_{p \xrightarrow{w}}(p')$. That is, $\mathit{read}^{m'}_{p \xrightarrow{w}}(p') = \mathit{read}^{m}_{p \xrightarrow{w}}(p')$. There are five cases:

- There exists $x \in \mathrm{dom}(m^{\pi}_{\pi'})$ such that $x \not\# m^{\pi}_{\pi'}(x)$. If $x \not\# m(x)$, then $x \not\# m'(x)$ since $m \subseteq m'$. If there exists $k \in \{1, \dots, n\}$ such that $x_{p_k} \not\# x_{p'_k}$, this does not change when extending m. Hence, $x \in \mathrm{dom}(m'^{\pi}_{\pi'})$ and $x \not\# m'^{\pi}_{\pi'}(x)$, *i.e.*, we have $w \vdash p \#^{m'} p'$.
- $o \neq o'$, which, clearly, does not depend on m. So, $w \vdash p \#^{m'} p'$.
- Likewise if $u = (x, c)$ and $u' = (x', c')$ with $c \neq c'$.
- $p_n, p'_n \in \mathcal{E}^{\mathcal{T}}$ and $|\chi_0(p_n)| \neq |\chi_0(p'_n)|$, which, again, does not depend on m. So, $w \vdash p \#^{m'} p'$.
- $p_n, p'_n \in \mathcal{E}^{\mathcal{T}}$ and there is $x \in \mathrm{dom}(m^{\pi}_{\pi'})$ such that $x \in \chi_0(p_n) \Leftrightarrow m^{\pi}_{\pi'}(x) \notin \chi_0(p'_n)$. If $x \in \mathrm{dom}(m)$ and as $m \subseteq m'$, we still have $x \in \chi_0(p_n) \Leftrightarrow m'(x) \notin \chi_0(p'_n)$. Likewise if there is a $k \in \{1, \dots, n\}$ such that $x_{p_k} \in \chi_0(p_n) \Leftrightarrow x_{p'_k} \notin \chi_0(p'_n)$. Therefore, we have again $w \vdash p \#^{m'} p'$.

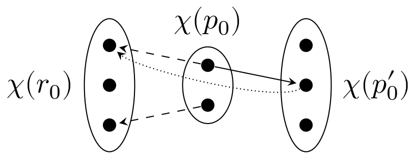
In every case, we obtain that $w \vdash p \#^{m'} p'$. □

Weak co-transitivity

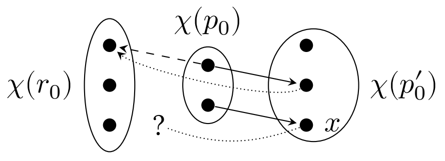
The next lemma states that if we can read a witness w of the *behavioral* apartness $p_0 \#^{m} p'_0$ from a third state r_0 via some matching $\mu : p_0 \leftrightarrow r_0$, then we can conclude that p_0 and r_0 are μ-apart or that p'_0 and r_0 are $(\mu \circ m^{-1})$-apart. However, when $p_0 \#^{m} p'_0$ is due to (constants), we need to extend the witness, due to (constants) in Definition 10.3.10. In this case, since x is active in p_n (as $u = (x, c)$), there must exist an x-spanning run $p_{n-1} \xrightarrow[u]{i_n/o} p_n \xrightarrow{w^x}$.[11] Hence, we actually "read" $w \cdot w^x$ from r_0, in order to ensure that an update (x', c') is present on the last transition of $\mathit{read}^{\mu}_{p_0 \xrightarrow{w}}(r_0)$.

$$\begin{aligned} & u = (x, c) \\ \wedge\ & u' = (x', c') \\ \wedge\ & c \neq c' \end{aligned} \qquad \text{(constants)}$$

11: This holds by definition of an observation tree, see Definition 10.3.1.



(a) Well-defined.



(b) Ill-defined: $(\mu \circ m^{-1})(x)$ has no value.

Figure 10.7: Visualizations of compositions $\mu \circ m^{-1}$ where m is drawn with solid lines, μ with dashed lines, and $\mu \circ m^{-1}$ with dotted lines.

Notice that the lemma requires that $\mathrm{dom}(m) \subseteq \mathrm{dom}(\mu)$ for the matching $\mu \circ m^{-1}$. See Figure 10.7 for illustrations of a well- and an ill-defined $\mu \circ m^{-1}$. Details are given in Section C.10.

Lemma 10.3.13 (Weak co-transitivity). *Let $p_0, p'_0, r_0 \in Q^{\mathcal{T}}$, $m : p_0 \leftrightarrow p'_0$ and $\mu : p_0 \leftrightarrow r_0$ be two matchings such that $\mathrm{dom}(m) \subseteq \mathrm{dom}(\mu)$. Let $w = i_1 \cdots i_n$ be a witness of the behavioral apartness $p_0 \#^m p'_0$ and $\mathrm{read}^m_{p_0 \xrightarrow{w} p_n}(p'_0) = p'_0 \xrightarrow{w'} p'_n$. Let w^x be defined as follows:*

- *if $p_0 \#^m p'_0$ due to (constants), w^x is a word such that $p_{n-1} \xrightarrow{i_n} p_n \xrightarrow{w^x}$ is x-spanning,*
- *otherwise, $w^x = \varepsilon$.*

If $\mathrm{read}^\mu_{p_0 \xrightarrow{w \cdot w^x}}(r_0) \in \mathit{runs}(\mathcal{T})$ with $r_n \in \mathcal{E}^{\mathcal{T}}$, then $p_0 \#^\mu r_0$ or $p'_0 \#^{\mu \circ m^{-1}} r_0$.

Soundness

Finally, we argue that the definition of apartness is reasonable: when $p \#^m p'$, then at least one of the following must hold:

- $f(p) \neq f(p')$, *i. e.*, the two states are really distinct, or
- $g(x) \neq g(m(x))$ for some x, *i. e.*, the renaming of the timers is erroneous.

A proof is given in Section C.11.

Theorem 10.3.14 (Soundness). *Let $\mathcal{T}$ be an observation tree for an s-learnable MMT $\mathcal{M}$ with the functional simulation $\langle f, g \rangle$, $p, p' \in Q^{\mathcal{T}}$, and $m : p \leftrightarrow p'$ be a matching. If $p \#^m p'$, then*

- *$f(p) \neq f(p')$, or*
- *there is $x \in \mathrm{dom}(m)$ such that $g(x) \neq g(m(x))$.*

10.4. Learning algorithm

We now describe our learning algorithm for MMTs, called $L^\#_{\mathrm{MMT}}$. Let $\mathcal{M}$ be the hidden MMT we want to learn. Again, we require that $\mathcal{M}$ is sound and complete.

In $L^\#_{\mathrm{MMT}}$, similarly to what is done in $L^\#$ [Vaa+22], the learner constructs

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness" an observation tree $\mathcal{T}$ for $\mathcal{M}$ by asking $\mathbf{OQ}^{\mathrm{s}}$, $\mathbf{WQ}^{\mathrm{s}}$, and $\mathbf{EQ}^{\mathrm{s}}$. Eventually, an *hypothesis* $\mathcal{H}$ is constructed, *i.e.*, a machine the learner thinks to be correct. If $\mathcal{H}$ and $\mathcal{M}$ are equivalent, the algorithm stops. Otherwise, the tree is extended by some *counterexample* provided by the teacher and then refined until the conditions are satisfied once more. We highlight that the functional simulation $\langle f, g\rangle : \mathcal{T} \to \mathcal{M}$ is unknown to the learner. Nevertheless, it can obtain some information about it via the notion of apartness introduced in Section 10.3.2.

We first explain how to extend the tree using symbolic queries. Then, as we did in Section 3.3.1 for $L^{\#}$, we define the basis and the frontier and explain how to compute these sets while enforcing some properties in Section 10.4.2. These properties allow us to construct the hypothesis $\mathcal{H}$, as explained in Section 10.4.3. The main loop of $L^{\#}_{\mathrm{MMT}}$ is given in Section 10.4.4, while Section 10.4.5 focuses on counterexample processing. Finally, a complete example is found in Section 10.4.6. The following theorem gives the complexity of $L^{\#}_{\mathrm{MMT}}$. Section C.15 gives a proof (that requires the next sections). We highlight that if $|X^{\mathcal{M}}|$ is fixed, then $L^{\#}_{\mathrm{MMT}}$ is polynomial in $|Q^{\mathcal{M}}|$, $|I|$, and the length of the longest counterexample.

Theorem 10.4.1. *Let $\mathcal{M}$ be an s-learnable MMT and ζ be the length of the longest counterexample. Then,*

- *the $L^{\#}_{\mathrm{MMT}}$ algorithm eventually terminates and returns an MMT $\mathcal{N}$ such that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ and whose size is polynomial in $|Q^{\mathcal{M}}|$ and factorial in $|X^{\mathcal{M}}|$, and*
- *in time and number of $\mathbf{OQ}^{\mathrm{s}}, \mathbf{WQ}^{\mathrm{s}}, \mathbf{EQ}^{\mathrm{s}}$ polynomial in $|Q^{\mathcal{M}}|$, $|I|$, and ζ, and factorial in $|X^{\mathcal{M}}|$.*

10.4.1. Using symbolic queries to extend the tree

In this section, we explain how to use symbolic output and wait queries to extend the tree, while maintaining the set of explored states. We first give a formal description before illustrating the procedure. Assume $\mathcal{T}$ is already an observation tree for $\mathcal{M}$ with the functional simulation $\langle f, g\rangle$. Let $q \in Q^{\mathcal{T}}$ and $w = i_1 \cdots i_n$ be the unique word such that $q_0^{\mathcal{T}} \xrightarrow{w} q \in \mathit{runs}(\mathcal{T})$. We want to create the outgoing transitions from q. Let $\mathtt{w} = \overline{w}$ be the symbolic word of w and

$$p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_n} p_n \in \mathit{runs}(\mathcal{T})$$

and

$$f(p_0) \xrightarrow{g(i_1)} \cdots \xrightarrow{g(i_n)} f(p_n) \in \mathit{runs}(\mathcal{M}),$$

with $p_0 = q_0^{\mathcal{T}}$ and $p_n = q$, be the concrete runs reading $\mathtt{w}$ in $\mathcal{T}$ and $\mathcal{M}$. As the run exists in $\mathcal{T}$, the corresponding run in $\mathcal{M}$ necessarily exists, by definition of $\langle f, g\rangle$.

First, we focus on creating the i-transition from q for an input $i \in I$. As $\mathcal{M}$ is complete, it follows that

$$q_0^{\mathcal{M}} \xrightarrow{g(i_1)\cdots g(i_n)} f(q) \xrightarrow{g(i)=i} \in \mathit{runs}(\mathcal{M}).$$

Moreover, i can be used as a symbolic symbol $\mathtt{i} = i$, by definition. Hence, $\mathbf{OQ^s}(\mathtt{w} \cdot \mathtt{i})$ returns a sequence of outputs $\omega \cdot o$ (with $\omega \in O^*$ and $o \in O$). We create a new state q' in $\mathcal{T}$ and define the transition $q \xrightarrow[\perp]{i/o} q'$ with a $\perp$-update as we do not know anything yet about the update of $f(q) \xrightarrow{i}$ in $\mathcal{M}$.

Creating the timeout-transitions from q requires more care. We first have to determine what is the set of enabled timers of $f(q)$. To do so, we ask a symbolic wait query $\mathbf{WQ^s}(\mathtt{w})$ that indicates which transitions of the run in $\mathcal{M}$ last (re)started a timer x that is enabled in $f(q)$. That is, for every pair (j, c) returned by the teacher, we know that for some timer x,

$$f(p_{j-1}) \xrightarrow[(x,c)]{g(i_j)} \cdots \xrightarrow{g(i_n)} f(p_n) \xrightarrow{\mathit{to}[x]}$$

is x-spanning in $\mathcal{M}$. So, for each such pair (j, c), we modify the j-th transition $p_{j-1} \xrightarrow[u]{i_j} p_j$ in $\mathcal{T}$ by replacing $u = \perp$ with $u = (y, c)$ where $y = x_{p_j}$ if $i_j \in I$, and $y = x'$ if $i_j = \mathit{to}[x']$. This j-th transition now (re)starts a timer y.

It remains to create the $\mathit{to}[y]$-transition from q to get that $y \in \chi_0^{\mathcal{T}}(q)$. By the wait query, we are sure that

$$q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} f(p_n) \xrightarrow{\mathit{to}[j]} \in \mathit{runs}(\mathcal{M})$$

and, so, $\mathbf{OQ^s}(\mathtt{w} \cdot \mathit{to}[j])$ necessarily returns a sequence of outputs $\omega \cdot o$ (again, $\omega \in O^*$ and $o \in O$). We can thus create a new transition $q \xrightarrow[\perp]{\mathit{to}[y]/o} q'$. By treating in this way all pairs (j, c) returned by the teacher, some updates may be changed, meaning that some of the traversed p_j in $\mathcal{T}$ may have new active timers.

$\langle f, g\rangle(\pi)$ is y-spanning $\Rightarrow$
$\exists x : \pi$ is x-spanning (FS5)
$\wedge\, g(x) = y$

It is not hard to see that $\mathcal{T}$ remains an observation tree for $\mathcal{M}$ with a functional simulation $\langle f', g'\rangle$ extending $\langle f, g\rangle$ by encompassing the new states and timers of $\mathcal{T}$. Indeed, we only create a transition $q \xrightarrow{i} q'$ when we are sure that $f'(q) \xrightarrow{g'(i)} f'(q')$ is defined; updates come from a wait query and (FS5) is satisfied. Finally, $\mathcal{T}$ is sound and all of its runs are feasible.

Definition 10.3.6. Let $q \in Q^{\mathcal{T}}$ and π be the unique run from $q_0^{\mathcal{T}}$ to q in $\mathcal{T}$. We say that q is *explored* if

$$|\chi_0^{\mathcal{T}}(q)| = |\chi_0^{\mathcal{M}}(f(q))|.$$

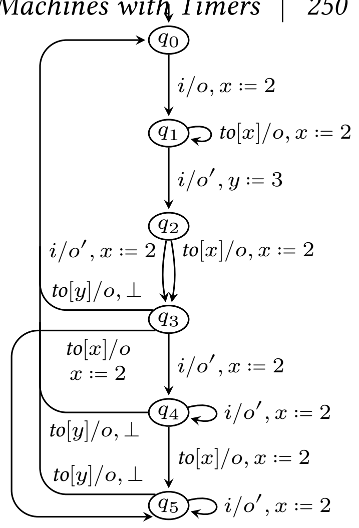
Define $\mathcal{E}^{\mathcal{T}}$ as the maximal set of explored states of $\mathcal{T}$ that induces a subtree containing $q_0^{\mathcal{T}}$, *i. e.*, $p \in \mathcal{E}^{\mathcal{T}}$ for all $p \xrightarrow{i} q$ with $q \in \mathcal{E}^{\mathcal{T}}$.

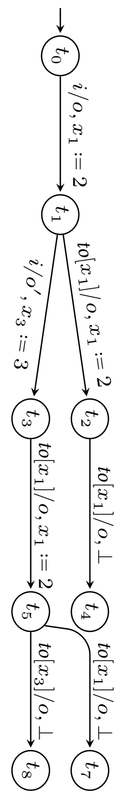
Let us now discuss the set $\mathcal{E}^{\mathcal{T}}$ of explored states. Clearly, as $\mathcal{M}$ is s-learnable, when we call $\mathbf{WQ^s}(\mathtt{w})$ and create the appropriate transitions from q, we have

$$|\chi_0^{\mathcal{T}}(q)| = |\chi_0^{\mathcal{M}}(f(q))|,$$

as required in Definition 10.3.6. Hence, the tree-shaped set $\mathcal{E}^{\mathcal{T}}$ is exactly composed of the initial state $q_0^{\mathcal{T}}$ (for which $\chi_0^{\mathcal{T}}(q_0^{\mathcal{T}}) = \emptyset$) and the states in which we performed a wait query. This means that when the outgoing transitions of a state q have been newly computed, we can add q to $\mathcal{E}^{\mathcal{T}}$.

Example 10.4.2. Let $\mathcal{M}$ be the s-learnable MMT of Figure 10.2, and $\mathcal{T}$ be the observation tree of Figure 10.5, except that $t_3 \xrightarrow{i} \notin$ *runs*$(\mathcal{T})$, *i.e.*, the subtree rooted at t_6 is not present in the tree. Both figures are given in the margin. We construct the missing subtree, via $\mathbf{OQ}^{\mathsf{s}}$ and $\mathbf{WQ}^{\mathsf{s}}$. Let $\mathcal{E}^{\mathcal{T}} = \{t_0, t_1, t_2, t_3, t_5\}$.
First, we create the $t_3 \xrightarrow{i}$ transition. Let $w = i \cdot i$ be the unique word such that $t_0 \xrightarrow{w} t_3$ and $\mathtt{w} = \overline{w} = i \cdot i$ be the corresponding symbolic word. As $\mathcal{T}$ is an observation tree for $\mathcal{M}$, if follows that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w} \cdot i} \in$ *runs*$(\mathcal{M})$. So, we can call $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w} \cdot i)$, which returns $o \cdot o' \cdot o'$. The last symbol o' must then be outputted by the new transition, *i.e.*, we create $t_3 \xrightarrow[\perp]{i/o'} t_6$. Indeed, recall that every transition has initially a $\perp$ update, which is changed when an update (x, c) is discovered.
We then perform a symbolic wait query in t_6, *i.e.*, call $\mathbf{WQ}^{\mathsf{s}}(\mathtt{w} \cdot i)$, which returns the set $\{(2, 3), (3, 2)\}$ meaning that the second transition of the run $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w} \cdot i}$ must (re)start a timer with the constant 3, and the third transition must also (re)start a timer but with the constant 2. So, the $\perp$ of the newly created transition is replaced by $(x_6, 2)$ (as the label of the transition is an input). It remains to create the *to*$[x_3]$- and *to*$[x_6]$-transitions by performing two symbolic output queries. We call $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w} \cdot i \cdot \mathit{to}[2])$ and $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w} \cdot i \cdot \mathit{to}[3])$ (we know that both words label runs of $\mathcal{M}$ by the symbolic wait query), and create the transitions. We thus obtain the tree of Figure 10.5. Finally, since $t_3 \in \mathcal{E}^{\mathcal{T}}$ and we discovered the set of enabled timers of t_6, we can add t_6 to $\mathcal{E}^{\mathcal{T}}$. Observe that $\mathcal{E}^{\mathcal{T}}$ still satisfies Definition 10.3.6.


q_0
$i/o, x := 2$
q_1
$\mathit{to}[x]/o, x := 2$
$i/o', y := 3$
q_2
$i/o', x := 2$
$\mathit{to}[x]/o, x := 2$
$\mathit{to}[y]/o, \perp$
q_3
$\mathit{to}[x]/o$
$x := 2$
$i/o', x := 2$
q_4
$i/o', x := 2$
$\mathit{to}[y]/o, \perp$
$\mathit{to}[x]/o, x := 2$
$\mathit{to}[y]/o, \perp$
q_5
$i/o', x := 2$


t_0
$i/o, x_1 := 2$
t_1
$i/o', x_3 := 3$
$\mathit{to}[x_1]/o, x_1 := 2$
t_3
t_2
$\mathit{to}[x_1]/o, x_1 := 2$
$\mathit{to}[x_1]/o, \perp$
t_5
t_4
$\mathit{to}[x_3]/o, \perp$
$\mathit{to}[x_1]/o, \perp$
t_8
t_7

From now on, we assume that a call to $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w} \cdot \mathtt{i})$ with $\mathtt{i} \in I$ automatically adds the corresponding transition to $\mathcal{T}$, and that a call to $\mathbf{WQ}^{\mathsf{s}}(\mathtt{w})$ automatically calls $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w} \cdot \mathit{to}[j])$, for every *to*$[j]$ deduced from the wait query, modifies updates accordingly, and adds the new explored states to $\mathcal{E}^{\mathcal{T}}$. That is, $\mathcal{E}^{\mathcal{T}}$ is exactly the set of states in which we performed a $\mathbf{WQ}^{\mathsf{s}}$. Moreover, to ease the writing in the learning algorithm, we let $\mathbf{OQ}^{\mathsf{s}}(q, i)$ denote $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w} \cdot \mathtt{i})$ and $\mathbf{WQ}^{\mathsf{s}}(q)$ denote $\mathbf{WQ}^{\mathsf{s}}(\mathtt{w})$ with $\mathtt{w}$ such that $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}} q \in$ *runs*$(\mathcal{T})$.

10.4.2. Basis and frontier

Let us now move towards constructing a hypothesis MMT $\mathcal{H}$ from $\mathcal{T}$. In short, we extend the tree such that some conditions are satisfied and we define a subset of $Q^{\mathcal{T}}$, called the *basis*, that forms the set of states of $\mathcal{H}$. Similar to $L^{\#}$ (see Section 3.3), we then "fold" the tree; that is, some transitions $q \to r$ must be redirected to some state p of the basis. In order to be able to construct a sound hypothesis, we impose that every active timer of r can be mapped to an active timer of p, and vice-versa. That is, we will ensure that r is mapped to a state p with the same number of active timers before constructing a hypothesis. Furthermore, to cover every active timer of the states, we use *maximal* matchings (introduced in Section 10.3.2). Let $\langle f, g \rangle : \mathcal{T} \to \mathcal{M}$ be a functional simulation.

In this section, we define the aforementioned subset of $\mathcal{T}$ and state the constraints that must be satisfied in order to construct $\mathcal{H}$. The construction is given in Section 10.4.3. As is done in $L^\#$, we split the states of $\mathcal{T}$ into three subsets:

- The *basis* $\mathcal{B}^\mathcal{T}$ is a subtree of $Q^\mathcal{T}$ such that $q_0^\mathcal{T} \in \mathcal{B}^\mathcal{T}$ and $p \#^m p'$ for any $p \neq p' \in \mathcal{B}^\mathcal{T}$ and maximal matching $m : p \leftrightarrow p'$. By Theorem 10.3.14, we thus know that $f(p) \neq f(p')$ or $g(x) \neq g(m(x))$ for some $x \in \mathsf{dom}(m)$. As we have this for every maximal m, we *conjecture* that $f(p) \neq f(p')$. We may be wrong, *i.e.*, $f(p) = f(p')$ but we need a matching we do not know yet, due to unknown active timers which will be discovered later.
- The *frontier* $\mathcal{F}^\mathcal{T} \subseteq Q^\mathcal{T}$ is the set of immediate non-basis successors of basis states, *i.e.*,

$$\mathcal{F}^\mathcal{T} = \{r \in Q^\mathcal{T} \setminus \mathcal{B}^\mathcal{T} \mid \exists p \in \mathcal{B}^\mathcal{T}, i \in A(\mathcal{T}) : p \xrightarrow{i} r\}.$$

  We say that $p \in \mathcal{B}^\mathcal{T}$ and $r \in \mathcal{F}^\mathcal{T}$ are *compatible* under a maximal matching m if $\neg(p \#^m r)$, *i.e.*, we cannot conjecture that $f(p) \neq f(r)$. We write $\mathit{compat}^\mathcal{T}(r)$ for the set of all such pairs (p, m):

$$\mathit{compat}^\mathcal{T}(r) = \{(p, m) \mid p \in \mathcal{B}^\mathcal{T} \wedge \neg(p \#^m r)\}.$$

- The remaining states $Q^\mathcal{T} \setminus (\mathcal{B}^\mathcal{T} \cup \mathcal{F}^\mathcal{T})$.

Theorem 10.3.14. Let $\mathcal{T}$ be an observation tree for an s-learnable MMT $\mathcal{M}$ with the functional simulation $\langle f, g\rangle$, $p, p' \in Q^\mathcal{T}$, and $m : p \leftrightarrow p'$ be a matching. If $p \#^m p'$, then

- $f(p) \neq f(p')$, or
- there is $x \in \mathsf{dom}(m)$ such that $g(x) \neq g(m(x))$.

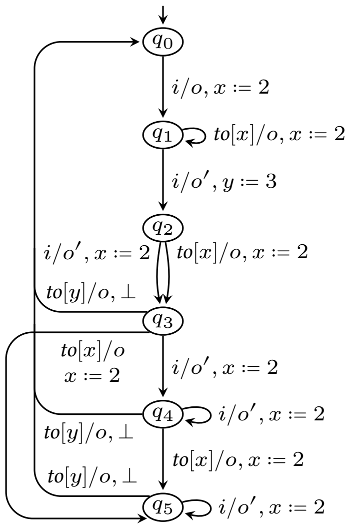
Example 10.4.3. Let the MMT of the teacher be the s-learnable MMT $\mathcal{M}$ of Figure 10.2 and $\mathcal{T}$ be the observation tree of Figure 10.5. Both figures are repeated in the margin. One can check that t_0, t_1, and t_3 are all pairwise apart under any maximal matching. We have

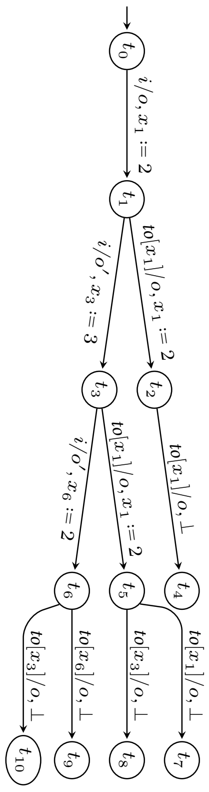
$$\begin{aligned} &\varepsilon \vdash t_0 \#^\emptyset t_1 && \varepsilon \vdash t_0 \#^\emptyset t_3 \\ &i \vdash t_1 \#^{x_1 \mapsto x_1} t_3 && i \vdash t_1 \#^{x_1 \mapsto x_3} t_3. \end{aligned}$$

Hence, t_0, t_1, and t_3 are all pairwise apart under any maximal matching. We can thus define $\mathcal{B}^\mathcal{T} = \{t_0, t_1, t_3\}$ and $\mathcal{F}^\mathcal{T} = \{t_2, t_5, t_6\}$. Let us compute $\mathit{compat}^\mathcal{T}(r)$ for each frontier state r. We have the following apartness pairs:

$$\begin{aligned} &\varepsilon \vdash t_0 \#^\emptyset t_2 && \neg(t_1 \#^{x_1 \mapsto x_1} t_2) \\ &\neg(t_3 \#^{x_1 \mapsto x_1} t_2) && \varepsilon \vdash t_3 \#^{x_3 \mapsto x_1} t_2 \\ &\varepsilon \vdash t_0 \#^\emptyset t_5 && \varepsilon \vdash t_1 \#^{x_1 \mapsto x_1} t_5 \\ &\varepsilon \vdash t_1 \#^{x_1 \mapsto x_3} t_5 && \varepsilon \vdash t_3 \#^{x_1 \mapsto x_1, x_3 \mapsto x_3} t_5 \\ &\varepsilon \vdash t_3 \#^{x_1 \mapsto x_3, x_3 \mapsto x_1} t_5 && \varepsilon \vdash t_0 \#^\emptyset t_6 \\ &\varepsilon \vdash t_1 \#^{x_1 \mapsto x_3} t_6 && \varepsilon \vdash t_1 \#^{x_1 \mapsto x_6} t_6 \\ &\varepsilon \vdash t_3 \#^{x_1 \mapsto x_3, x_3 \mapsto x_6} t_6 && \varepsilon \vdash t_3 \#^{x_1 \mapsto x_6, x_3 \mapsto x_3} t_6. \end{aligned}$$

Hence, $\mathit{compat}^\mathcal{T}(t_2) = \{(t_1, x_1 \mapsto x_1), (t_3, x_1 \mapsto x_1)\}$ while $\mathit{compat}^\mathcal{T}(t_5) = \mathit{compat}^\mathcal{T}(t_6) = \emptyset$.





Throughout the rest of this section, we explain how to extend $\mathcal{T}$ such that a sound and complete MMT can be constructed from $\mathcal{T}$. The following definition gives the requirements that must be satisfied before being able to construct this hypothesis.

Definition 10.4.4 (Requirements for a hypothesis)**.** In order to be able to construct a hypothesis from the observation tree, we will ensure that the following requirements are met:

Explored each basis and frontier state is explored, *i.e.*, $\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}} \subseteq \mathcal{E}^{\mathcal{T}}$, in order to discover timers as quickly as possible,

Complete the basis is *complete*, in the sense that $p \xrightarrow{i}$ is defined for every $i \in I \cup \mathit{TO}[\chi_0^{\mathcal{T}}(p)]$, and

Active timers for every $r \in \mathcal{F}^{\mathcal{T}}$, $\mathit{compat}^{\mathcal{T}}(r) \neq \emptyset$ and $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$.

Computing the basis and the frontier

We now explain how to compute $\mathcal{B}^{\mathcal{T}}$ and $\mathcal{F}^{\mathcal{T}}$ via calls to $\mathbf{OQ}^{\mathbf{s}}$ and $\mathbf{WQ}^{\mathbf{s}}$. We initialize the tree to only contain $q_0^{\mathcal{T}}$, *i.e.*, $\mathcal{B}^{\mathcal{T}} = \{q_0^{\mathcal{T}}\}$ and $\mathcal{F}^{\mathcal{T}} = \emptyset$. As $\chi^{\mathcal{M}}(q_0^{\mathcal{M}}) = \emptyset$, we know that $q_0^{\mathcal{T}}$ must be considered enabled and, so, $\mathcal{E}^{\mathcal{T}} = \{q_0^{\mathcal{T}}\}$, which is then initially tree-shaped.[12]

12: This is required in Definition 10.3.6.

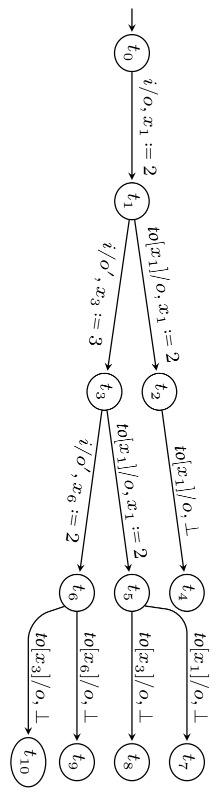
If $q \xrightarrow{i}$ is not defined for some $i \in I$ and $q \in \mathcal{B}^{\mathcal{T}}$, (*i.e.*, if $\mathcal{B}^{\mathcal{T}}$ is not complete), we call $\mathbf{OQ}^{\mathbf{s}}(q, i)$ to create the new state r, which is added to $\mathcal{F}^{\mathcal{T}}$. As we want $\mathcal{F}^{\mathcal{T}} \subseteq \mathcal{E}^{\mathcal{T}}$, we call $\mathbf{WQ}^{\mathbf{s}}(r)$. At some point during learning, we may discover that a frontier state r has an empty $\mathit{compat}^{\mathcal{T}}(r)$. That is, $p \#^{m} r$ for every $p \in \mathcal{B}^{\mathcal{T}}$ and maximal matching $m : p \leftrightarrow r$. We can thus *promote* r to $\mathcal{B}^{\mathcal{T}}$.[13] There may again be a missing transition, leading to a $\mathbf{OQ}^{\mathbf{s}}$, or a successor of r is not explored, requiring a $\mathbf{WQ}^{\mathbf{s}}$. As $r \in \mathcal{E}^{\mathcal{T}}$ and by the wait query performed to do so (see Section 10.4.1), it follows that $\delta(r, i)$ is already defined for every $i \in \mathit{TO}[\chi_0^{\mathcal{T}}(r)]$, *i.e.*, a missing transition must read an input (not a timeout).

13: As $r \in \mathcal{E}^{\mathcal{T}}$, we still satisfy the requirement *Explored* of Definition 10.4.4, and $\mathcal{B}^{\mathcal{T}}$ remains tree-shaped.

Example 10.4.5. Let us continue Example 10.4.3 (again, the tree is repeated in the margin). As $\mathit{compat}^{\mathcal{T}}(t_6) = \emptyset$, we promote t_6, *i.e.*, $\mathcal{B}^{\mathcal{T}} = \{t_0, t_1, t_3, t_6\}$. We call $\mathbf{WQ}^{\mathbf{s}}(t_9)$ and $\mathbf{WQ}^{\mathbf{s}}(t_{10})$, which yields $\chi(t_9) = \chi_0(t_9) = \{x_3\}$ and $\chi(t_{10}) = \emptyset$. Hence, $\mathcal{F}^{\mathcal{T}} = \{t_2, t_5, t_9, t_{10}\}$, and

$$\begin{aligned}
\mathit{compat}^{\mathcal{T}}(t_2) &= \{(t_1, x_1 \mapsto x_1), (t_3, x_1 \mapsto x_1)\}\\
\mathit{compat}^{\mathcal{T}}(t_5) &= \{(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)\}\\
\mathit{compat}^{\mathcal{T}}(t_9) &= \emptyset\\
\mathit{compat}^{\mathcal{T}}(t_{10}) &= \{(t_0, \emptyset)\}.
\end{aligned}$$

First, the last line holds as $\chi_0^{\mathcal{T}}(t_{10}) = \emptyset$, $t_{10} \in \mathcal{E}^{\mathcal{T}}$. So, the only possible compatible state is t_0. As t_{10} does not have any outgoing transition, we cannot obtain $t_0 \#^{\emptyset} t_{10}$. Second, $\mathit{compat}^{\mathcal{T}}(t_9)$ is empty, since the only states with the same number of enabled timers are t_1 and t_3 (as $\chi_0(t_1) = \chi_0(t_3) = \{x_1\}$) but any matching $x_1 \mapsto x_3$ is invalid. So, $t_1 \#^{x_1 \mapsto x_3} t_9$ and $t_3 \#^{x_1 \mapsto x_3} t_9$. Finally, $t_3 \#^{x_3 \mapsto x_3} t_9$ as $x_3 \notin \chi_0(t_3)$ but $x_3 \in \chi_0(t_9)$.



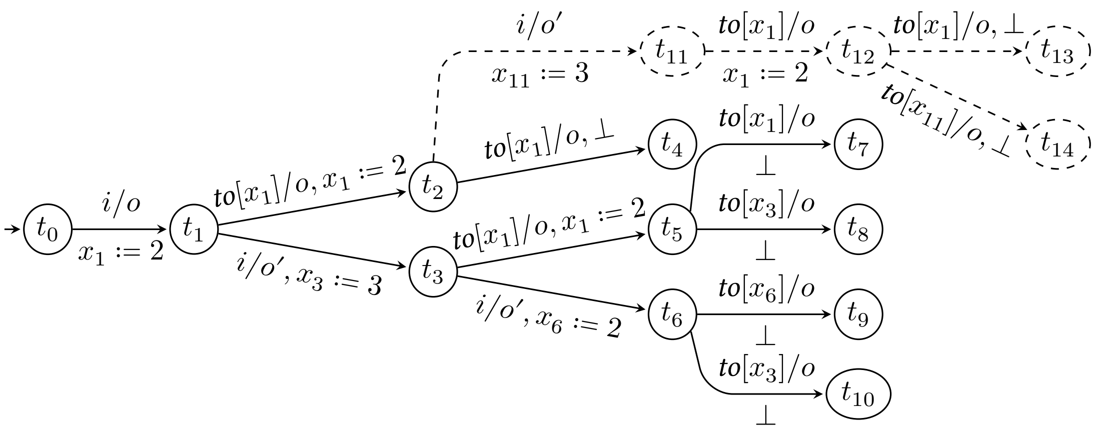
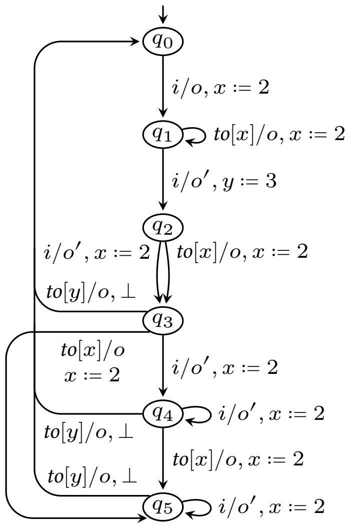


Figure 10.8: Extension of the observation tree of Figure 10.5 obtained by calling $\mathit{replay}_\pi^{x_1 \mapsto x_1}(t_2)$, where $\pi = t_1 \xrightarrow{i \cdot to[x_1] \cdot to[x_3]}$. New states and transitions are highlight with dashed lines.

Replaying a run

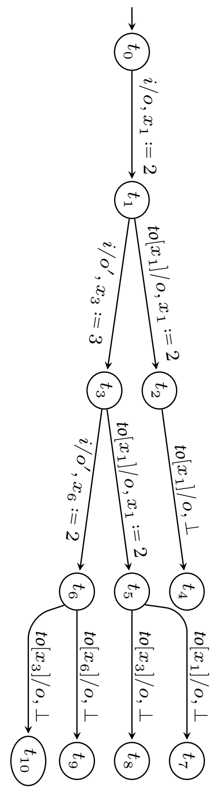
By the previous section, both requirements *Explored* and *Complete* of Definition 10.4.4 are satisfied. It remains the last item *Active timers*, *i.e.*, to ensure that $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $r \in \mathcal{F}^{\mathcal{T}}$ and $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$. To ease the writing, assume that $|\chi^{\mathcal{T}}(p)| > |\chi^{\mathcal{T}}(r)|$. The other case can easily be obtained with the same approach. There must exist a timer x that is active in p but not used in m (*i.e.*, $x \in \chi^{\mathcal{T}}(p) \setminus \mathrm{dom}(m)$). By definition of $\mathcal{T}$, there exists a word w ending with $to[x]$ such that $\pi = p \xrightarrow{w} \in \mathit{runs}(\mathcal{T})$. That is, π shows that x eventually times out. We want to *replay* π from r, *i.e.*, add new nodes and transitions to $\mathcal{T}$ to replicate the behavior of π from r. As soon as we find a new timer in r or that $p \#^m r$, we can stop replaying the run. We introduce the function replay_π^m via an example. Technical details and a pseudo-code are given in Section C.12.



Example 10.4.6. Let $\mathcal{M}$ be the MMT of Figure 10.2 and $\mathcal{T}$ be the observation tree of Figure 10.5. Both figures are repeated in the margin. Let

$$\pi = t_1 \xrightarrow{i/o'} t_3 \xrightarrow{to[x_1]/o} t_5 \xrightarrow{to[x_3]/o} t_8.$$

We replay π from t_2, using the matching $m : t_1 \leftrightarrow t_2$ such that $m = x_1 \mapsto x_1$, *i.e.*, we call $\mathit{replay}_\pi^m(t_2)$. First, we check whether we already have $t_1 \#^m t_2$, in which case it is unnecessary to extend the tree. As it is not the case, we start processing the run π from t_2. We first perform a wait query in t_2, which has no effect here, as $t_2 \in \mathcal{E}^{\mathcal{T}}$ (as $t_2 \xrightarrow{to[x_1]} \in \mathit{runs}(\mathcal{T})$). The first symbol of π is an input, so we call $\mathbf{OQ^s}(t_2, i)$ and define the transition $t_2 \xrightarrow{i/o'} t_{11}$. As we did not find a new active timer in t_2 nor $t_1 \#^m t_2$, we proceed with the next symbol.
We reproduce $to[x_1]$ from t_{11} by first calling $\mathbf{WQ^s}(t_{11})$, which does not bring anything new. Observe that $x_1 \in \mathrm{dom}(m)$ and $to[m(x_1)] = to[x_1] \in \chi_0(t_{11})$. So, we can take the transition $t_{11} \xrightarrow{to[x_1]} t_{12}$ and process the last symbol



to$[x_3]$ from t_{12}. We do $\mathbf{WQ}^{\mathsf{s}}(t_{12})$ and learn that $t_2 \xrightarrow[(x_{11},3)]{} t_{11} \xrightarrow[(x_1,2)]{} t_{12}$. We still have no new active timer in t_2, nor $t_1 \#^m t_2$. Since $x_3 \in \chi_0(t_5)$ is a fresh timer started by the first transition of π and $x_{11} \in \chi_0(t_{12})$ is a fresh timer started on the corresponding transition in the new run, we take the transition $t_{12} \to t_{14}$. As the run π is completely replayed, the algorithm returns DONE. We obtain the tree of Figure 10.8, where the new states and transitions are drawn with dashed lines.
Had we found a new active timer (resp. $t_1 \#^m t_2$), replay^m_π would have returned ACTIVE (resp. APART). For instance, starting from Figure 10.5 again, $\mathit{replay}^{m'}_{t_3 \xrightarrow{i} t_6 \xrightarrow{to[x_6]}}(t_2)$ (with m' any maximal matching) also adds t_{11} and t_{12} (due to the output and wait queries) but returns APART (as $|\chi_0(t_6)| \neq |\chi_0(t_{11})|$).

Observe that $\mathit{replay}^m_\pi(q)$ takes the same transitions as $\mathit{read}^m_\pi(q)$ (if the run exists) but also extends the tree. The next proposition (proved in Section C.12) highlights important properties of this replay algorithm. We conclude that we can ensure the last requirement of Definition 10.4.4, *i. e.*, that $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $r \in \mathcal{F}^{\mathcal{T}}$ and $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$ by applying

- $\mathit{replay}^m_{p \xrightarrow{w}}(r)$ with w a word ending in *to*$[x]$ for a timer $x \in \chi^{\mathcal{T}}(p) \setminus \mathrm{dom}(m)$, if $|\chi^{\mathcal{T}}(p)| > |\chi^{\mathcal{T}}(r)|$, or
- $\mathit{replay}^{m^{-1}}_{r \xrightarrow{w}}(p)$ with w a word ending in *to*$[x]$ for a timer $x \in \chi^{\mathcal{T}}(r) \setminus \mathrm{dom}(m^{-1})$, if $|\chi^{\mathcal{T}}(p)| < |\chi^{\mathcal{T}}(r)|$.

The following proposition gives general properties of the replay algorithm (*i. e.*, not only for the context of identifying new active timers we studied here).

Proposition 10.4.7. *Let $p_0, p_0' \in Q^{\mathcal{T}}$, $m : p_0 \leftrightarrow p_0'$ be a maximal matching, and $\pi = p_0 \xrightarrow{w} \in \mathit{runs}(\mathcal{T})$. Then,*

- *if $\mathit{replay}^m_\pi(p_0')$ returns* DONE*, then $\mathit{read}^m_\pi(p_0')$ is now a run of $\mathcal{T}$.*
- *$\mathit{replay}^m_\pi(p_0')$ returns* APART *or* ACTIVE *if $|\chi^{\mathcal{T}}(p_0)| > |\chi^{\mathcal{T}}(p_0')|$ and w ends with to$[x]$ for some $x \in \chi^{\mathcal{T}}(p_0) \setminus \mathrm{dom}(m)$.*

Definition 10.4.4. In order to be able to construct a hypothesis from the observation tree, we will ensure that the following requirements are met:

Explored each basis and frontier state is explored, *i. e.*, $\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}} \subseteq \mathcal{E}^{\mathcal{T}}$, in order to discover timers as quickly as possible,
Complete the basis is *complete*, in the sense that $p \xrightarrow{i}$ is defined for every $i \in I \cup \mathit{TO}[\chi_0^{\mathcal{T}}(p)]$, and
Active timers for every $r \in \mathcal{F}^{\mathcal{T}}$, $\mathit{compat}^{\mathcal{T}}(r) \neq \emptyset$ and $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$.

Minimization of $\mathit{compat}^{\mathcal{T}}(r)$

Intuitively, the hypothesis construction (which is presented in the next section) will, for every frontier state r, arbitrarily select a pair (p, m) in $\mathit{compat}^{\mathcal{T}}(r)$ and "redirect" the transition leading to r to go in p. This is possible as soon as the requirements of Definition 10.4.4 are satisfied, in particular, when $\mathit{compat}^{\mathcal{T}}(r)$ is not empty for every frontier state r. Since these requirements do not impose that each compatible set contains a single element, there are multiple potential hypothesis that can be constructed from the tree. If we reduce the size of each compatible set, we reduce the number of possible hypotheses, which in turn reduces the overall number of equivalence queries we need. Hence, we now focus on minimizing each compatible set. More precisely, we extend the tree in order to apply the weak co-transitivity lemma (see Lemma 10.3.13) as much as possible.

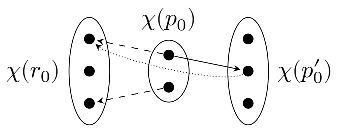
Lemma 10.3.13. Let p_0, p_0', r_0 be three states of $\mathcal{T}$, $m : p_0 \leftrightarrow p_0'$ and $\mu : p_0 \leftrightarrow r_0$ be two matchings such that $\mathrm{dom}(m)$ is a subset of $\mathrm{dom}(\mu)$, $w = i_1 \cdots i_n$ be a witness of the behavioral apartness $p_0 \#^m p_0'$, and $\mathit{read}^m_{p_0 \xrightarrow{w} p_n}(p_0') = p_0' \xrightarrow{w'} p_n'$. Moreover, let w^x be a word such that $p_{n-1} \xrightarrow{i_n} p_n \xrightarrow{w^x}$ is x-spanning if $p_0 \#^m p_0'$ due to (constants), or be ε otherwise. If $\mathit{read}^\mu_{p_0 \xrightarrow{w \cdot w^x}}(r_0)$ is a run of $\mathcal{T}$ with $r_n \in \mathcal{E}^{\mathcal{T}}$, then $p_0 \#^\mu r_0$ or $p_0' \#^{\mu \circ m^{-1}} r_0$.

Let $r \in \mathcal{F}^{\mathcal{T}}$, and (p, μ) and (p', μ') be two pairs in $compat^{\mathcal{T}}(r)$ with $p \neq p'$ and maximal matchings $\mu : p \leftrightarrow r$ and $\mu' : p' \leftrightarrow r$. These matchings are necessarily valid by definition of a compatible set. We also assume that

$$|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)| = |\chi^{\mathcal{T}}(p')|.$$

This can be obtained by replaying runs, as explained above.

As $p, p' \in \mathcal{B}^{\mathcal{T}}$, it must be that $p \#^m p'$ for any maximal matching $m : p \leftrightarrow p'$. In particular, take $m : p \leftrightarrow p'$ such that $m = \mu'^{-1} \circ \mu$. Notice that $\text{dom}(m) \subseteq \text{dom}(\mu)$ and $\mu' = \mu \circ m^{-1}$ (see Figure 10.7, which is repeated in the margin, for a visualization). It always exists and is unique as the three states have the same number of active timers. There are two cases:



- either any witness $w \vdash p \#^m p'$ is such that the apartness is structural, in which case we cannot apply Lemma 10.3.13, or
- there is a witness $w \vdash p \#^m p'$ where the apartness is behavioral. In that case, let also w^x be as described in Lemma 10.3.13. We then replay the run $p \xrightarrow{w \cdot w^x}$ from r using μ. We have three cases:
  - $replay^{\mu}_{p \xrightarrow{w \cdot w^x}}(r) = \text{APART}$, meaning that $p \#^{\mu} r$. Then, (p, μ) is no longer in $compat^{\mathcal{T}}(r)$.
  - $replay^{\mu}_{p \xrightarrow{w \cdot w^x}}(r) = \text{ACTIVE}$, in which case we discovered a new active timer in r. Hence, we now have that $|\chi^{\mathcal{T}}(p)| \neq |\chi^{\mathcal{T}}(r)|$ and we can again replay some runs, as explained above, to obtain the equality again, or that p and r are not compatible anymore.
  - $replay^{\mu}_{p \xrightarrow{w \cdot w^x}}(r) = \text{DONE}$, meaning that we could fully replay $p \xrightarrow{w \cdot w^x}$ and thus did not obtain $p \#^{\mu} r$. By Lemma 10.3.13, it follows that $p' \#^{\mu'} r$.

  Hence, it is sufficient to call $replay^{\mu}_{p \xrightarrow{w \cdot w^x}}(r)$ when $w \vdash p \#^{\mu'^{-1} \circ \mu} p'$ is behavioral.

We highlight that we cannot always obtain that each compatible set contains a single element, as Lemma 10.3.13 cannot be applied when the considered apartness pairs are structural.

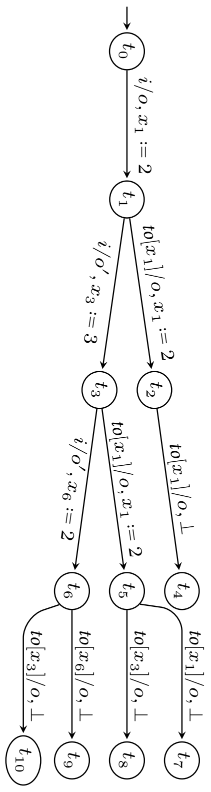
Example 10.4.8. Let $\mathcal{T}$ be the observation tree of Figure 10.5, which is repeated in the margin. As explained in Example 10.4.3, we have

$$compat^{\mathcal{T}}(t_2) = \{(t_1, x_1 \mapsto x_1), (t_3, x_1 \mapsto x_1)\}.$$

Let us extend the tree in order to apply weak co-transitivity to deduce that $t_1 \#^{x_1 \mapsto x_1} t_2$ or $t_1 \#^{x_1 \mapsto x_1} t_3$. We have that $i \vdash t_1 \#^{x_1 \mapsto x_1} t_3$ due to (constants). Hence, we replay the run

$$\pi = t_1 \xrightarrow{i \cdot to[x_1] \cdot to[x_3]}$$

from t_2 using the matching $x_1 \mapsto x_1$ (*i.e.*, we have $w^x = to[x_1] \cdot to[x_3]$). That is, we call $replay^{x_1 \mapsto x_1}_{\pi}(t_2)$. The computations are given in Example 10.4.6 and the resulting tree in Figure 10.8. Recall that the function returned DONE.



So, $\neg(t_1 \#^{m_1 \mapsto x_1} t_2)$. By Lemma 10.3.13, it must be that $t_3 \#^{x_1 \mapsto x_1} t_2$. It is indeed the case as $i \vdash t_3 \#^{x_1 \mapsto x_1} t_2$ by (constants). Hence, we now have

$$compat^{\mathcal{T}}(t_2) = \{(t_1, x_1 \mapsto x_1)\}.$$

$$\begin{aligned} & u = (x, c) \\ \wedge\, & u' = (x', c') \\ \wedge\, & c \neq c' \end{aligned} \qquad \text{(constants)}$$

10.4.3. Constructing a hypothesis

In this section, we explain how to construct a hypothesis $\mathcal{H}$ from $\mathcal{T}$ such that $Q^{\mathcal{H}} = \mathcal{B}^{\mathcal{T}}$. We assume that the observation tree respects the requirements of Definition 10.4.4. The idea is to pick a $(p, m) \in compat^{\mathcal{T}}(r)$ for each frontier state r. Then, the unique transition $q \xrightarrow{i} r$ in $\mathcal{T}$ becomes $q \xrightarrow{i} p$ in $\mathcal{H}$. We also *globally* rename the timers according to m.

Definition 10.4.4. In order to be able to construct a hypothesis from the observation tree, we will ensure that the following requirements are met:

Explored each basis and frontier state is explored, *i. e.*, $\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}} \subseteq \mathcal{E}^{\mathcal{T}}$, in order to discover timers as quickly as possible,

Complete the basis is *complete*, in the sense that $p \xrightarrow{i}$ is defined for every $i \in I \cup TO[\chi_0^{\mathcal{T}}(p)]$, and

Active timers for every $r \in \mathcal{F}^{\mathcal{T}}$, $compat^{\mathcal{T}}(r) \neq \emptyset$ and $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $(p, m) \in compat^{\mathcal{T}}(r)$.

Let

$$X^{\mathcal{B}^{\mathcal{T}}} = \bigcup_{q \in \mathcal{B}^{\mathcal{T}}} \chi^{\mathcal{T}}(q)$$

be the set of timers used within the basis, and

$$X^{\mathcal{F}^{\mathcal{T}}} = \bigcup_{r \in \mathcal{F}^{\mathcal{T}}} \chi^{\mathcal{T}}(r)$$

be the set of timers used within the frontier. We construct a (total) function $\mathfrak{h} : \mathcal{F}^{\mathcal{T}} \to \mathcal{B}^{\mathcal{T}}$ that dictates how to fold the frontier states, and an equivalence relation $\equiv\ \subseteq (X^{\mathcal{B}^{\mathcal{T}}} \cup X^{\mathcal{F}^{\mathcal{T}}}) \times (X^{\mathcal{B}^{\mathcal{T}}} \cup X^{\mathcal{F}^{\mathcal{T}}})$ to know how to rename the timers.

- The relation $\equiv$ is initialized with $x \equiv x$ for each $x \in X^{\mathcal{B}^{\mathcal{T}}}$, *i. e.*, $\equiv$ is reflexive and each timer of the basis has its own class.
- Then, we add the timers used in the frontier. For every $r \in \mathcal{F}^{\mathcal{T}}$, arbitrarily select a $(p, m) \in compat^{\mathcal{T}}(r)$. We then define $\mathfrak{h}(r) = p$ (*i. e.*, we send r to p) and add $x \equiv m(x)$ for every $x \in \mathrm{dom}(m)$.
- We then compute the transitive and symmetric closure of $\equiv$.

Finally, we check whether $\equiv$ satisfies that two apart timers are not put together, *i. e.*, $\forall x \neq y : x \equiv y \Rightarrow \neg(x \# y)$. If it does not hold, $\equiv$ is discarded and we restart by selecting a different (p, m) for some frontier state r.

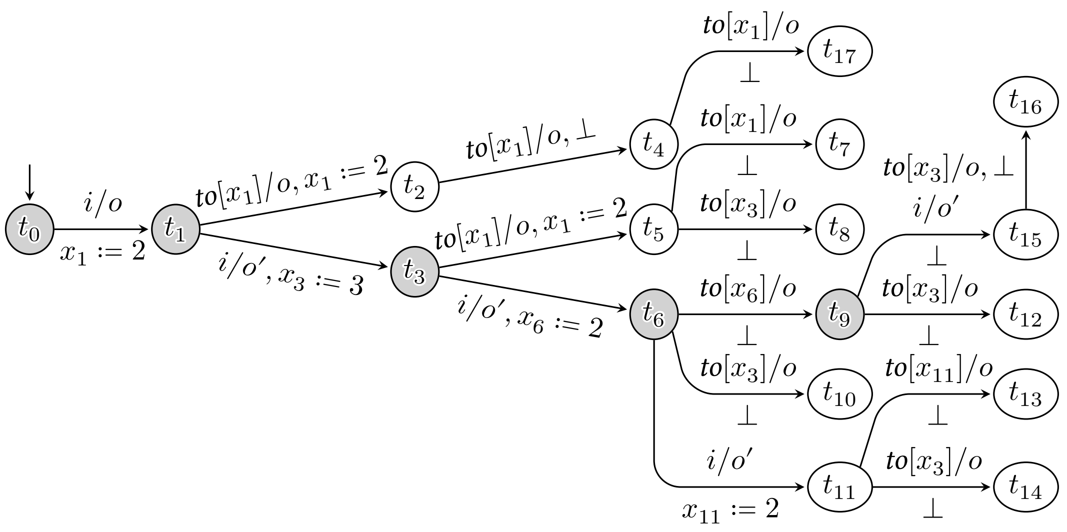
Let us now define the MMT constructed from $\mathfrak{h}$ and $\equiv$. We denote by $[\![x]\!]_{\equiv}$ the equivalence class of the timer x for the relation $\equiv$. We lift $[\![\cdot]\!]_{\equiv}$ to actions i and updates u:

$$[\![i]\!]_{\equiv} = \begin{cases} i & \text{if } i \in I \\ to[[\![x]\!]_{\equiv}] & \text{if } i = to[x] \text{ with } x \in X^{\mathcal{B}^{\mathcal{T}}} \cup X^{\mathcal{F}^{\mathcal{T}}} \end{cases}$$

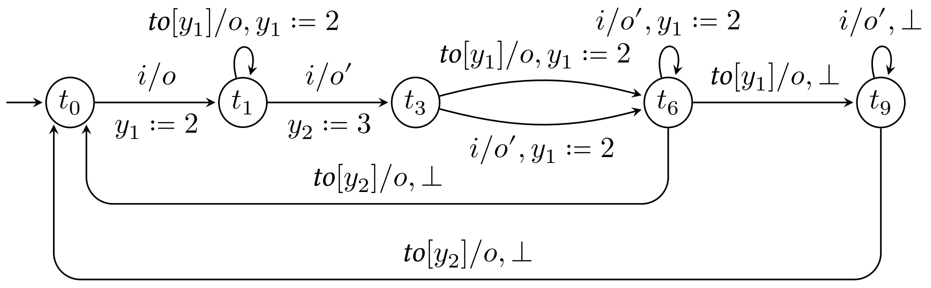
and

$$[\![u]\!]_{\equiv} = \begin{cases} \bot & \text{if } u = \bot \\ ([\![x]\!]_{\equiv}, c) & \text{if } u = (x, c) \text{ with } x \in X^{\mathcal{B}^{\mathcal{T}}} \cup X^{\mathcal{F}^{\mathcal{T}}}. \end{cases}$$

Definition 10.4.9 (MMT hypothesis). Let $\mathfrak{h} : \mathcal{F}^{\mathcal{T}} \to \mathcal{B}^{\mathcal{T}}$ and $\equiv\ \subseteq (X^{\mathcal{B}^{\mathcal{T}}} \cup X^{\mathcal{F}^{\mathcal{T}}}) \times (X^{\mathcal{B}^{\mathcal{T}}} \cup X^{\mathcal{F}^{\mathcal{T}}})$ be as constructed above. We define an MMT $\mathcal{N} =$



(a) The observation tree.



(b) The hypothesis.

Figure 10.9: On top, an observation tree from which the hypothesis MMT at the bottom is constructed, with $y_1 = [\![x_1]\!]_\equiv$ and $y_2 = [\![x_3]\!]_\equiv$. Basis states are highlighted with a gray background.

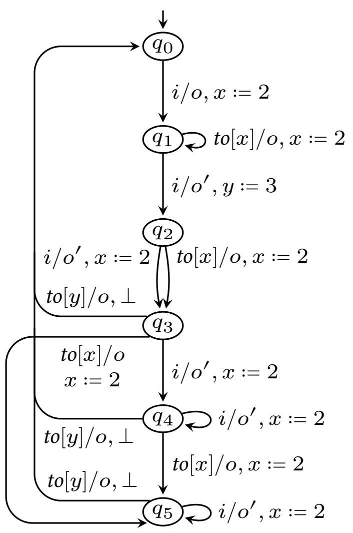
$(I, O, X^\mathcal{N}, Q^\mathcal{N}, q_0^\mathcal{N}, \chi^\mathcal{N}, \delta^\mathcal{N})$ where:

- $X^\mathcal{N} = \bigcup_{q \in \mathcal{B}^\mathcal{T}} \chi^\mathcal{T}(q)$,
- $Q^\mathcal{N} = \mathcal{B}^\mathcal{T}$ with $q_0^\mathcal{N} = q_0^\mathcal{T}$,
- $\chi^\mathcal{N}(q) = \{[\![x]\!]_\equiv \mid x \in \chi^\mathcal{T}(q)\}$, and
- the transition function $\delta^\mathcal{N}$ is defined as follows. Let $q \xrightarrow[u]{i/o} q'$ be a transition of $\mathcal{T}$ with $q \in \mathcal{B}^\mathcal{T}$. We set

$$\delta^\mathcal{N}(q, [\![i]\!]_\equiv) = \begin{cases} (q', o, [\![u]\!]_\equiv) & \text{if } q' \in \mathcal{B}^\mathcal{T} \\ (\mathfrak{h}(q'), o, [\![u]\!]_\equiv) & \text{if } q' \in \mathcal{F}^\mathcal{T}. \end{cases}$$

It is not hard to see that $\mathcal{N}$ is sound and complete.

Example 10.4.10. Let $\mathcal{M}$ be the s-learnable MMT of Figure 10.2 (which is repeated in the margin) and $\mathcal{T}$ be the observation tree of Figure 10.9a. Basis states are highlighted with a gray background, *i.e.*, $\mathcal{B}^\mathcal{T} = \{t_0, t_1, t_3, t_6, t_9\}$ and $\mathcal{F}^\mathcal{T} = \{t_2, t_5, t_{10}, t_{11}, t_{12}, t_{15}\}$. Moreover, we have the following com-



patible sets:

$$\begin{aligned} \mathit{compat}^{\mathcal{T}}(t_2) &= \{(t_1, x_1 \mapsto x_1)\} \\ \mathit{compat}^{\mathcal{T}}(t_{10}) = \mathit{compat}^{\mathcal{T}}(t_{12}) &= \{(t_0, \emptyset)\} \\ \mathit{compat}^{\mathcal{T}}(t_5) &= \{(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)\} \\ \mathit{compat}^{\mathcal{T}}(t_{15}) &= \{(t_9, x_3 \mapsto x_3)\} \\ \mathit{compat}^{\mathcal{T}}(t_{11}) &= \{(t_6, x_6 \mapsto x_{11}, x_3 \mapsto x_3)\}. \end{aligned}$$

We construct $\mathcal{H}$ with $Q^{\mathcal{H}} = \mathcal{B}^{\mathcal{T}}$. While defining the transitions $q \to q'$ is easy when $q, q' \in \mathcal{B}^{\mathcal{T}}$, we have to redirect the transition to some basis state when $q' \in \mathcal{F}^{\mathcal{T}}$. To do so, we first define a map $\mathfrak{h} : \mathcal{F}^{\mathcal{T}} \to \mathcal{B}^{\mathcal{T}}$, and an equivalence relation $\equiv$ over the set of active timers of the basis and the frontier. For each $r \in \mathcal{F}^{\mathcal{T}}$, we pick $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$, define $\mathfrak{h}(r) = p$, and add $x \equiv m(x)$ for every $x \in \mathsf{dom}(m)$ (and compute the symmetric and transitive closure of $\equiv$). Here, we obtain

$$\begin{aligned} \mathfrak{h}(t_2) &= t_1 & \mathfrak{h}(t_5) = \mathfrak{h}(t_{11}) &= t_6 \\ \mathfrak{h}(t_{10}) = \mathfrak{h}(t_{12}) &= t_0 & \mathfrak{h}(t_{15}) &= t_9 \end{aligned}$$

and

$$x_1 \equiv x_6 \equiv x_{11} \qquad\qquad x_3 \equiv x_3.$$

We check whether we have $x \equiv y$ and $x \mathrel{\#} y$, in which case, we restart again by picking some different (p, m). Here, this does not hold and we construct $\mathcal{H}$ by copying the transitions starting from a basis state (while folding the tree when required), except that a timer x is replaced by its equivalence class $[\![x]\!]_{\equiv}$. Figure 10.9b gives the resulting $\mathcal{H}$. Observe that x_1, x_6, and x_{11} are all renamed into y_1, *i.e.*, the three different timers of $\mathcal{T}_7$ become a single timer.

We highlight that it is *not* always possible to construct $\equiv$ such that $\neg(x \mathrel{\#} y)$ for every $x \equiv y$. In that case, we instead construct a *generalized* MMT, in which every transition can arbitrarily rename the active timers. The size of that generalized MMT is also $|\mathcal{B}^{\mathcal{T}}|$. From the generalized MMT, a classical MMT can be constructed of size $n! \cdot |\mathcal{B}^{\mathcal{T}}|$, with $n = \max_{p \in \mathcal{B}^{\mathcal{T}}} |\chi^{\mathcal{T}}(p)|$. Details are given in Section C.13. We observed on practical examples that a valid $\equiv$ can often be constructed. Section C.13 also illustrates a case where a $\equiv$ does not exist.

Definition 10.4.4. In order to be able to construct a hypothesis from the observation tree, we will ensure that the following requirements are met:

Explored each basis and frontier state is explored, *i. e.*, $\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}} \subseteq \mathcal{E}^{\mathcal{T}}$, in order to discover timers as quickly as possible,

Complete the basis is *complete*, in the sense that $p \xrightarrow{i}$ is defined for every $i \in I \cup TO[\chi_0^{\mathcal{T}}(p)]$, and

Active timers for every $r \in \mathcal{F}^{\mathcal{T}}$, $\mathit{compat}^{\mathcal{T}}(r) \neq \emptyset$ and $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$.

10.4.4. Main loop

We now give the main loop of $L^{\#}_{\mathrm{MMT}}$. We initialize $\mathcal{T}$ to only contain $t_0^{\mathcal{T}}$, $\mathcal{B}^{\mathcal{T}} = \mathcal{E}^{\mathcal{T}} = \{q_0^{\mathcal{T}}\}$, and $\mathcal{F}^{\mathcal{T}} = \emptyset$. The main loop is split into two parts:

Refinement loop The *refinement loop* extends the tree to obtain the requirements of Definition 10.4.4, by performing the following operations, in this order, until no more changes are possible:

Seismic If we discover a new active timer in a basis state, then it may be that $\neg(q \mathrel{\#^m} q')$ for some $q, q' \in \mathcal{B}^{\mathcal{T}}$ and maximal m, due to the new timer. Indeed, if q and q' have the same number of timers before discovering the new active timers, then there may be some maximal matchings between the two states for which we do not have a witness of the apartness yet. To avoid this, we reset the basis back to $\{q_0^{\mathcal{T}}\}$, as soon as a new timer is found, without removing states from $\mathcal{T}$. Notice that we do not remove any state or transition from the tree.

Promotion If $\mathit{compat}^{\mathcal{T}}(r)$ is empty for some frontier state r, then we know that $q \#^m r$ for every $q \in \mathcal{B}^{\mathcal{T}}$ and maximal matching $m : q \leftrightarrow r$. Hence, we promote r to the basis.

Completion If an i-transition is missing from some basis state p, we complete the basis with that transition. Recall that it is sufficient to only check for i that are inputs, as every basis state is explored.

Active timers We ensure that p and r have the same number of active timers for every $(p, \cdot) \in \mathit{compat}^{\mathcal{T}}(r)$.

WCT where **WCT** stands for Weak Co-Transitivity. As explained above, we minimize each compatible set by extending the tree to leverage Lemma 10.3.13 as much as possible.

Lemma 10.3.13. Let p_0, p_0', r_0 be three states of $\mathcal{T}$, $m : p_0 \leftrightarrow p_0'$ and $\mu : p_0 \leftrightarrow r_0$ be two matchings such that $\mathrm{dom}(m)$ is a subset of $\mathrm{dom}(\mu)$, $w = i_1 \cdots i_n$ be a witness of the behavioral apartness $p_0 \#^m p_0'$, and $\mathit{read}^m_{p_0 \xrightarrow{w} p_n}(p_0') = p_0' \xrightarrow{w'} p_n'$. Moreover, let w^x be a word such that $p_{n-1} \xrightarrow{i_n} p_n \xrightarrow{w^x}$ is x-spanning if $p_0 \#^m p_0'$ due to (constants), or be ε otherwise. If $\mathit{read}^\mu_{p_0 \xrightarrow{w \cdot w^x}}(r_0)$ is a run of $\mathcal{T}$ with $r_n \in \mathcal{E}^{\mathcal{T}}$, then $p_0 \#^\mu r_0$ or $p_0' \#^{\mu \circ m^{-1}} r_0$.

Hypothesis and equivalence Once the refinement loop no longer modifies $\mathcal{T}$ (*i.e.*, the requirements of Definition 10.4.4 are all satisfied), we can construct a hypothesis $\mathcal{H}$ from $\mathcal{T}$ and perform a symbolic equivalence query $\mathbf{EQ}^{\mathsf{s}}(\mathcal{H})$. If the teacher answers **yes**, we then return $\mathcal{H}$. Otherwise, a symbolic counterexample $\mathtt{w}$ is provided and can be used to extend $\mathcal{T}$ (as we explain in the next section), before refining it again.

A pseudo-code is given in Algorithm 10.1.

10.4.5. Counterexample processing

Let $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ be a counterexample returned by a call to $\mathbf{EQ}^{\mathsf{s}}(\mathcal{H})$ (see Definition 10.2.3). We process $\mathtt{w}$ by extending the tree to learn new apartness pair(s) or active timer(s). We want to obtain that (p, m) is no longer in $\mathit{compat}^{\mathcal{T}}(r)$ for some $r \in \mathcal{F}^{\mathcal{T}}$ such that (p, m) was selected to construct $\mathcal{H}$, or that a new timer is found in a basis state.

Observe that if we add $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}} q$ to the tree, then it must be that $q \notin \mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}$, due to how $\mathcal{H}$ is constructed. First, by construction, it is impossible to have a mistake with the transitions that remain within the basis. Second, the selected $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$ is such that $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ and $|\chi_0^{\mathcal{T}}(p)| = |\chi_0^{\mathcal{T}}(r)|$.[14] So, the counterexample shows a mistake due to a (potentially missing) transition $q \xrightarrow{i}$ with $q \notin \mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}$.

14: Due to the fact that the tree satisfies the requirements of Definition 10.4.4.

As $\mathtt{w}$ may be very long, we seek a sufficient prefix, by adding each $\mathtt{i}_k$ one by one while performing $\mathbf{WQ}^{\mathsf{s}}$ in each traversed state. A prefix $\mathtt{w}' \cdot \mathtt{i}$ of $\mathtt{w}$ is eventually found such that one of the following cases holds:

- a new timer is discovered in $\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}$,
- for some $r \in \mathcal{F}^{\mathcal{T}}$, we obtain $p \#^m r$ with (p, m) the selected pair for $\mathcal{H}$,
- $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}'} q \xrightarrow{\mathtt{i}} \in \mathit{runs}(\mathcal{T})$ and $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}'} q' \xrightarrow{\mathtt{i}} \notin \mathit{runs}(\mathcal{H})$, or the other direction,[15]

15: In this case $\mathtt{i} = \mathit{to}[j]$ for some j.

Algorithm 10.1: Overall $L^{\#}_{\text{MMT}}$ algorithm.

```
1:  Initialize $\mathcal{T}$ with $\mathcal{B}^{\mathcal{T}} = \{q_0^{\mathcal{T}}\}$ and $\mathcal{F}^{\mathcal{T}} = \emptyset$
2:  while true do
3:    while $\mathcal{T}$ is changed do                                                      ▷ Refinement loop
4:      if number of active timers of a basis state has changed then                          ▷ Seismic
5:        $\mathcal{B}^{\mathcal{T}} \leftarrow \{q_0^{\mathcal{T}}\}$ and $\mathcal{F}^{\mathcal{T}} \leftarrow \{r \mid \exists q_0^{\mathcal{T}} \xrightarrow{i} r\}$
6:      else if $\exists r \in \mathcal{F}^{\mathcal{T}}$ such that $compat^{\mathcal{T}}(r) = \emptyset$ then   ▷ Promotion
7:        $\mathcal{B}^{\mathcal{T}} \leftarrow \mathcal{B}^{\mathcal{T}} \cup \{r\}$
8:        $\mathcal{F}^{\mathcal{T}} \leftarrow \mathcal{F}^{\mathcal{T}} \setminus \{r\}$
9:        for all $r \xrightarrow{i} r'$ do
10:         WQ$^{\mathsf{s}}(r')$ and $\mathcal{F}^{\mathcal{T}} \leftarrow \mathcal{F}^{\mathcal{T}} \cup \{r'\}$
11:     else if $\exists p \in \mathcal{B}^{\mathcal{T}}, i \in I$ such that $p \xrightarrow{i} \notin runs(\mathcal{T})$ then   ▷ Completion
12:       OQ$^{\mathsf{s}}(p, i)$
13:       Let $r$ be such that $p \xrightarrow{i} r$
14:       WQ$^{\mathsf{s}}(r)$
15:       $\mathcal{F}^{\mathcal{T}} \leftarrow \mathcal{F}^{\mathcal{T}} \cup \{r \mid p \xrightarrow{i} r\}$
16:     else if $\exists r \in \mathcal{F}^{\mathcal{T}}, (p, m) \in compat^{\mathcal{T}}(r)$ such that $|\chi^{\mathcal{T}}(p)| > |\chi^{\mathcal{T}}(r)|$ then   ▷ Active timers
17:       Let $w$ be such that $p \xrightarrow{w \cdot to[x]}$ for some $x \in \chi^{\mathcal{T}}(p) \setminus \mathrm{dom}(m)$
18:       $replay^{m}_{p \xrightarrow{w \cdot to[x]}}(r)$
19:     else if $\exists r \in \mathcal{F}^{\mathcal{T}}, (p, m) \in compat^{\mathcal{T}}(r)$ such that $|\chi^{\mathcal{T}}(p)| < |\chi^{\mathcal{T}}(r)|$ then
20:       Let $w$ be such that $r \xrightarrow{w \cdot to[x]}$ for some $x \in \chi^{\mathcal{T}}(r) \setminus \mathrm{dom}(m^{-1})$
21:       $replay^{m^{-1}}_{r \xrightarrow{w \cdot to[x]}}(p)$
22:     else if $\exists r \in \mathcal{F}^{\mathcal{T}}, (p, \mu), (p', \mu') \in compat^{\mathcal{T}}(r), w \vdash p \,\#^{\mu'^{-1} \circ \mu}\, p'$ is behavioral then   ▷ WCT
23:       $replay^{\mu}_{p \xrightarrow{w \cdot w^x}}(r)$ with $w^x$ as described in Lemma 10.3.13
24:   ▷ Hypothesis construction, once $\mathcal{B}^{\mathcal{T}}$ and $\mathcal{F}^{\mathcal{T}}$ are stabilized
25:   $\mathcal{H} \leftarrow$ ConstructHypothesis
26:   $v \leftarrow$ EQ$^{\mathsf{s}}(\mathcal{H})$
27:   if $v =$ yes then return $\mathcal{H}$ else ProcCounterEx$(v)$
```

- $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}'} q \xrightarrow[u]{\mathtt{i}/o} \in runs(\mathcal{T})$, $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}'} q' \xrightarrow[u']{\mathtt{i}/o'} \in runs(\mathcal{H})$, and $o \neq o'$, or $u = (x, c), u' = (x', c')$ with $c \neq c'$.

In the first two cases, we already obtain our goal and we can stop processing the counterexample. For the third and four cases, it must be that $q \notin \mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}$, as said above. Let v be $\mathtt{w}'$ in the third case, and $\mathtt{w}' \cdot \mathtt{i}$ in the fourth case. We replay a part of v from some state in $\mathcal{T}$.

Let $r_1 \in \mathcal{F}^{\mathcal{T}}$ and v_1, v_1' be such that $q_0^{\mathcal{T}} \xrightarrow{v_1} r_1 \xrightarrow{v_1'}$, $v_1' \neq \varepsilon$, and $\overline{v_1 \cdot v_1'} = \mathsf{v}$. That is, we split the run of $\mathcal{T}$ reading v into the part that leads to a frontier state and its suffix. Moreover, let $(p_1, m_1) \in compat^{\mathcal{T}}(r_1)$ be the pair selected for $\mathcal{H}$. In order to try to get a new apartness pair, we replay v_1' from p_1 using m_1^{-1} as the matching. In other words, we call $replay^{m_1^{-1}}_{r_1 \xrightarrow{v_1'}}(p_1)$ which can return three values:

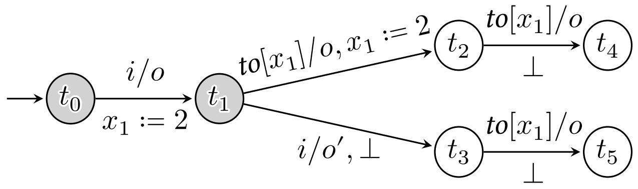


Figure 10.10: The observation tree $\mathcal{T}_3$. Basis states are highlighted with a gray background.

- APART, meaning $p_1 \#^{m_1} r_1$, *i.e.*, $(p_1, m_1) \notin \mathit{compat}^{\mathcal{T}}(r)$ and we stop.
- ACTIVE, meaning we discovered a new active timer in $p_1 \in \mathcal{B}^{\mathcal{T}}$, which is a seismic event and we stop.
- DONE, meaning $p_1 \xrightarrow{v_1'} \in \mathit{runs}(\mathcal{T})$ by Proposition 10.4.7. We keep processing the counterexample by applying the same idea: let $r_2 \in \mathcal{F}^{\mathcal{T}}$, $v_1' = v_2 \cdot v_2'$ be such that $p_1 \xrightarrow{v_2} r_2 \xrightarrow{v_2'}$, and $(p_2, m_2) \in \mathit{compat}^{\mathcal{T}}(r_2)$, and call $\mathit{replay}^{m_2}_{r_2 \xrightarrow{v_2'}}(p_2)$, leading to a similar case distinction.

Proposition 10.4.7. Let $p_0, p_0' \in Q^{\mathcal{T}}$, $m : p_0 \leftrightarrow p_0'$ be a maximal matching, and $\pi = p_0 \xrightarrow{w} \in \mathit{runs}(\mathcal{T})$. Then,

- if $\mathit{replay}^{m}_{\pi}(p_0')$ returns DONE, then $\mathit{read}^{m}_{\pi}(p_0')$ is now a run of $\mathcal{T}$.
- $\mathit{replay}^{m}_{\pi}(p_0')$ returns APART or ACTIVE if $|\chi^{\mathcal{T}}(p_0)| > |\chi^{\mathcal{T}}(p_0')|$ and w ends with $to[x]$ for some $x \in \chi^{\mathcal{T}}(p_0) \setminus \mathrm{dom}(m)$.

By the next lemma, we always eventually learn something new by processing a counterexample. A proof is given in Section C.14 while examples are provided in the next section.

Proposition 10.4.11. *When processing a counterexample, we eventually find a j such that $\mathit{replay}^{m_j^{-1}}_{r_j \xrightarrow{v_j'}}(p_j)$ returns* APART *or* ACTIVE.

10.4.6. Complete example

Finally, we perform a complete run of $L^{\#}_{\mathrm{MMT}}$, using the s-learnable MMT of Figure 10.2 as the MMT $\mathcal{M}$ of the teacher. For the sake of this example, we numerate each observation tree, obtained by applying some modifications, starting with $\mathcal{T}_1$. Likewise, we write $\mathcal{B}^{\mathcal{T}_j}, \mathcal{F}^{\mathcal{T}_j}$, *etc.*

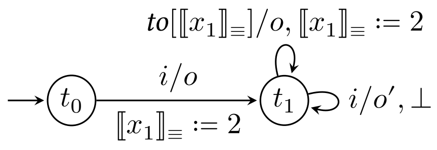
Initially, we have $Q^{\mathcal{T}_1} = \mathcal{B}^{\mathcal{T}_1} = \{t_0\}$ and $\mathcal{F}^{\mathcal{T}_1} = \emptyset$. We thus need to apply **Completion** and perform a symbolic output query to add the transition $t_0 \xrightarrow[\perp]{i/o} t_1$. Moreover, we do a symbolic wait query and learn that the transition must start the timer x_1 to the constant 2, *i.e.*, we have $t_0 \xrightarrow[(x_1,2)]{i/o} t_1 \xrightarrow[\perp]{to[x_1]/o} t_2$ and $\mathcal{F}^{\mathcal{T}_2} = \{t_1\}$. Since the number of enabled timers in t_0 and t_1 are different, we have $\mathit{compat}^{\mathcal{T}_2}(t_1) = \emptyset$, allowing us to apply **Promotion** and add t_1 to the basis. We immediately perform a wait query in t_2 and obtain

$$t_1 \xrightarrow[(x_1,2)]{to[x_1]/o} t_2 \xrightarrow[\perp]{to[x_1]/o} t_4.$$

Moreover, we apply **Completion** over t_1 and i to obtain

$$t_1 \xrightarrow[\perp]{i/o'} t_3 \xrightarrow[\perp]{to[x_1]/o} t_5.$$

The resulting observation tree $\mathcal{T}_3$ is given in Figure 10.10. We have the following



$to[[\![x_1]\!]_\equiv]/o, [\![x_1]\!]_\equiv := 2$
i/o
$[\![x_1]\!]_\equiv := 2$
t_0
t_1
$i/o', \bot$

Figure 10.11: The hypothesis MMT constructed from $\mathcal{T}_3$.

apartness pairs:

$$\begin{array}{lll} \varepsilon \vdash t_0 \#^{\emptyset} t_1 & \varepsilon \vdash t_0 \#^{\emptyset} t_2 & \varepsilon \vdash t_0 \#^{\emptyset} t_3. \\ \neg(t_1 \#^{x_1 \mapsto x_1} t_2) & \neg(t_1 \#^{x_1 \mapsto x_1} t_3). & \end{array}$$

The pairs of the first line are all due to (sizes). Hence, we can define $\mathcal{B}^{\mathcal{T}} = \{t_0, t_1\}$ and $\mathcal{F}^{\mathcal{T}} = \{t_2, t_3\}$. Moreover,

$p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge |\chi_0(p_n)| \neq |\chi_0(p'_n)|$ (sizes)

$$\mathit{compat}^{\mathcal{T}}(t_2) = \mathit{compat}^{\mathcal{T}}(t_3) = \{(t_1, x_1 \mapsto x_1)\}.$$

We thus satisfy, for each frontier state r, that $\mathit{compat}^{\mathcal{T}}(r) \neq \emptyset$ and p and r have the same number of active timers for every $(p, \cdot) \in \mathit{compat}^{\mathcal{T}}(r)$. One can check that $\mathcal{B}^{\mathcal{T}} = \{t_0, t_1\}$ is the maximal basis of $\mathcal{T}$. Then, $\mathcal{F}^{\mathcal{T}} = \{t_2, t_3\}$ and we have $\mathit{compat}^{\mathcal{T}}(t_2) = \mathit{compat}^{\mathcal{T}}(t_3) = \{(t_1, x_1 \mapsto x_1)\}$. We thus satisfy the required constraints to construct a hypothesis.

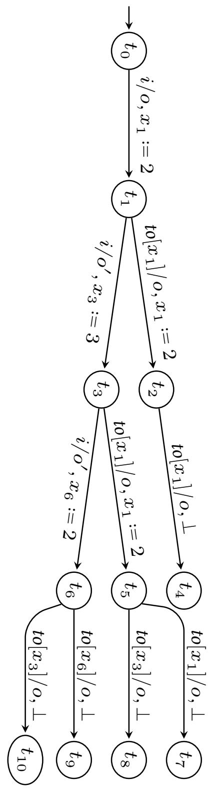
So, we can compute a map $\mathfrak{h} : \{t_2, t_3\} \to \{t_0, t_1\}$ and a relation $\equiv \ \subseteq \{x_1\} \times \{x_1\}$ such that:

$$\mathfrak{h}(t_2) = \mathfrak{h}(t_3) = t_1 \qquad \text{and} \qquad x_1 \equiv x_1.$$

We then construct the hypothesis $\mathcal{H}_1$ (given in Figure 10.11) and ask a symbolic equivalence query which returns $i \cdot i \cdot \mathit{to}[1] \cdot \mathit{to}[2] \cdot i \cdot i$.

Let us process that counterexample. First, we add w to $\mathcal{T}$ symbol by symbol. Observe that $q_0^{\mathcal{T}} \xrightarrow{i \cdot i \cdot \mathit{to}[1]} t_5$ is already a run of $\mathcal{T}_3$ but t_5 is not yet explored. So, we perform a wait query in t_5 which returns $\{(2, 3), (3, 2)\}$, *i. e.*, the transition $t_1 \xrightarrow{i} t_3$ starts a fresh timer x_3 at constant 3 and $t_3 \xrightarrow{\mathit{to}[x_1]} t_5$ restarts x_1 at constant 2. We thus obtain that $\chi_0(t_2) = \{x_1\}$ and $\chi_0(t_5) = \{x_1, x_3\}$, meaning that $\mathit{to}[x_1] \vdash t_1 \#^{x_1 \mapsto x_1} t_3$ by (sizes). That is, we immediately obtain that $(t_1, x_1 \mapsto x_1)$ is no longer compatible with t_3. Observe that we only added $i \cdot i \cdot \mathit{to}[1] \cdot \mathit{to}[2]$ to $\mathcal{T}$, *i. e.*, a proper prefix of w. Moreover, t_3 can be promoted, as $t_1 \#^{x_1 \mapsto x_1} t_3$ (by (sizes)) and $t_1 \#^{x_1 \mapsto x_3} t_3$ (the matching is invalid). Hence, we apply **Promotion** and the subsequent **Completion** to obtain the observation tree $\mathcal{T}_4$ given in Figure 10.5 (repeated in the margin), with $\mathcal{B}^{\mathcal{T}_4} = \{t_0, t_1, t_3\}$ and $\mathcal{F}^{\mathcal{T}_4} = \{t_2, t_5, t_6\}$. We have

$$\mathit{compat}^{\mathcal{T}_4}(t_2) = \{(t_1, x_1 \mapsto x_1), (t_3, x_1 \mapsto x_1)\}$$
$$\mathit{compat}^{\mathcal{T}_4}(t_5) = \mathit{compat}^{\mathcal{T}_4}(t_6) = \emptyset.$$



Both t_5 and t_6 can be promoted to the basis. Say that we apply **Promotion** on t_6, followed by a **Completion** to add the missing $t_6 \xrightarrow{i}$ transition. We obtain the observation tree $\mathcal{T}_5$ drawn with solid and dashed lines in Figure 10.12. We

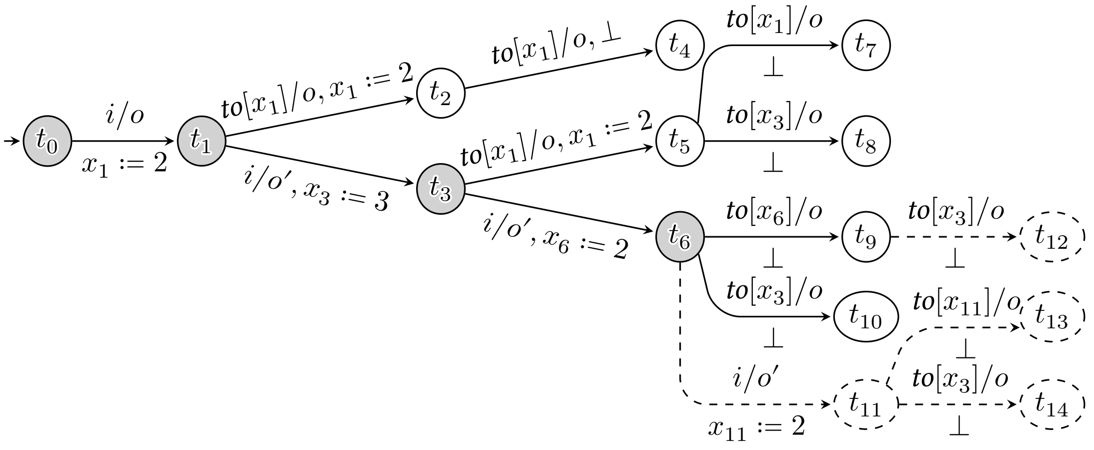


Figure 10.12: The observation tree $\mathcal{T}_5$. Newly added states and transitions are drawn with dashed lines.

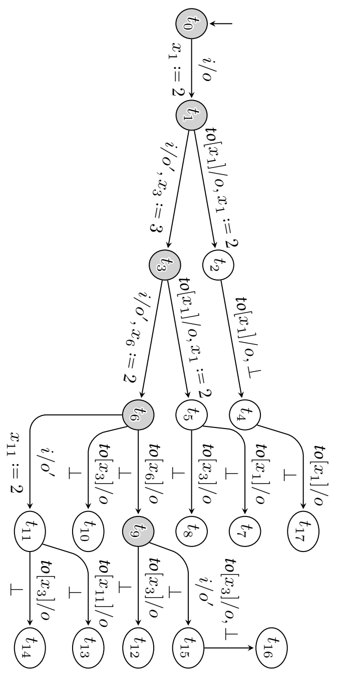
have

$$\begin{aligned}
compat^{\mathcal{T}_5}(t_2) &= \{(t_1, x_1 \mapsto x_1), (t_3, x_1 \mapsto x_1)\}\\
compat^{\mathcal{T}_5}(t_9) &= \emptyset\\
compat^{\mathcal{T}_5}(t_5) &= \{(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)\}\\
compat^{\mathcal{T}_5}(t_{10}) &= \{(t_0, \emptyset)\}\\
compat^{\mathcal{T}_5}(t_{11}) &= \{(t_6, x_6 \mapsto x_{11}, x_3 \mapsto x_3)\}.
\end{aligned}$$

Hence, we can apply **Promotion** on t_9 and **Completion** as needed. The resulting tree $\mathcal{T}_6$ is shown in Figure 10.13. We have

$$\begin{aligned}
compat^{\mathcal{T}_6}(t_2) &= \{(t_1, x_1 \mapsto x_1), (t_3, x_1 \mapsto x_1)\}\\
compat^{\mathcal{T}_6}(t_{10}) &= \{(t_0, \emptyset)\}\\
compat^{\mathcal{T}_6}(t_5) &= \{(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)\}\\
compat^{\mathcal{T}_6}(t_{12}) &= \{(t_0, \emptyset)\}\\
compat^{\mathcal{T}_6}(t_{11}) &= \{(t_6, x_6 \mapsto x_{11}, x_3 \mapsto x_3)\}\\
compat^{\mathcal{T}_6}(t_{15}) &= \{(t_9, x_3 \mapsto x_3)\}.
\end{aligned}$$

This time, we cannot apply **Promotion**. However, observe that $(t_3, x_1 \mapsto x_1) \in compat^{\mathcal{T}_6}(t_2)$ but $\chi^{\mathcal{T}_6}(t_3) = \{x_1, x_3\}$ while $\chi^{\mathcal{T}_6}(t_2) = \{x_1\}$. That is, we can apply **Active timers**. Let $\pi = t_3 \xrightarrow{to[x_1]} t_5 \xrightarrow{to[x_3]}$ be a run ending with $to[x_3]$. We replay π from t_2 using m, *i.e.*, call $replay^m_\pi(t_2)$. That algorithm performs a wait query in t_4 and discovers that $\chi_0^{\mathcal{T}_6}(t_4) = \{x_1\}$. Since $\chi_0^{\mathcal{T}_6}(t_5) = \{x_1, x_3\}$, we immediately obtain that $t_3 \#^{x_1 \mapsto x_1} t_2$ and we do not need to keep replaying the run. The resulting tree $\mathcal{T}_7$ is given in Figure 10.9a and is repeated in the



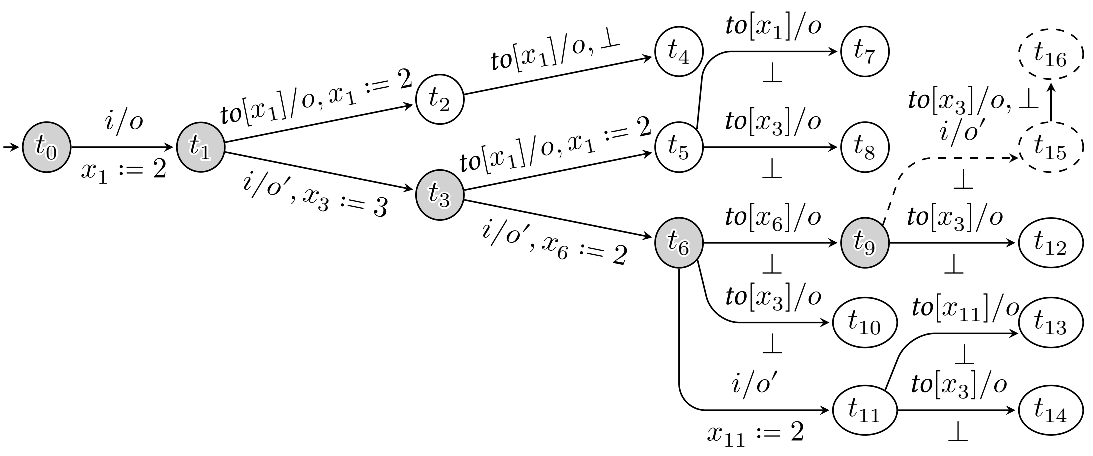
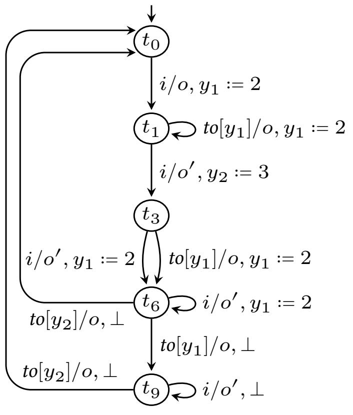


Figure 10.13: The observation tree $\mathcal{T}_6$. Newly added states and transitions are drawn with dashed lines.

margin. We have

$$\begin{aligned}
compat^{\mathcal{T}_6}(t_2) &= \{(t_1, x_1 \mapsto x_1)\} \\
compat^{\mathcal{T}_6}(t_{10}) = compat^{\mathcal{T}_6}(t_{12}) &= \{(t_0, \emptyset)\} \\
compat^{\mathcal{T}_6}(t_5) &= \{(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)\} \\
compat^{\mathcal{T}_6}(t_{15}) &= \{(t_9, x_3 \mapsto x_3)\} \\
compat^{\mathcal{T}_6}(t_{11}) &= \{(t_6, x_6 \mapsto x_{11}, x_3 \mapsto x_3)\}.
\end{aligned}$$

The refinement loop stops (as none of the operations can be applied). We thus construct the hypothesis given in Figure 10.9b (repeated in the margin), as done in Example 10.4.10. By asking a symbolic equivalence query, we obtain that $\mathcal{H}_2 \overset{\text{sym}}{\approx} \mathcal{M}$ and return $\mathcal{H}_2$.



Observe that $\mathcal{H}_2$ is almost isomorphic to $\mathcal{M}$: the states q_4 and q_5 of $\mathcal{M}$ are merged into a single state t_9.

To conclude, we recall Theorem 10.4.1, stating that $L^{\#}_{\text{MMT}}$ eventually finishes and its complexity. A proof is provided in Section C.15.

Theorem 10.4.1. *Let $\mathcal{M}$ be an s-learnable MMT and ζ be the length of the longest counterexample. Then,*

- *the $L^{\#}_{\text{MMT}}$ algorithm eventually terminates and returns an MMT $\mathcal{N}$ such that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ and whose size is polynomial in $|Q^{\mathcal{M}}|$ and factorial in $|X^{\mathcal{M}}|$, and*
- *in time and number of* $\mathbf{OQ}^{\mathsf{s}}, \mathbf{WQ}^{\mathsf{s}}, \mathbf{EQ}^{\mathsf{s}}$ *polynomial in $|Q^{\mathcal{M}}|$, $|I|$, and ζ, and factorial in $|X^{\mathcal{M}}|$.*

10.5. Implementation and experiments

In this section, we discuss our implementation and experimental results. More details will be available in the PhD Thesis of Bharat Garhewal.

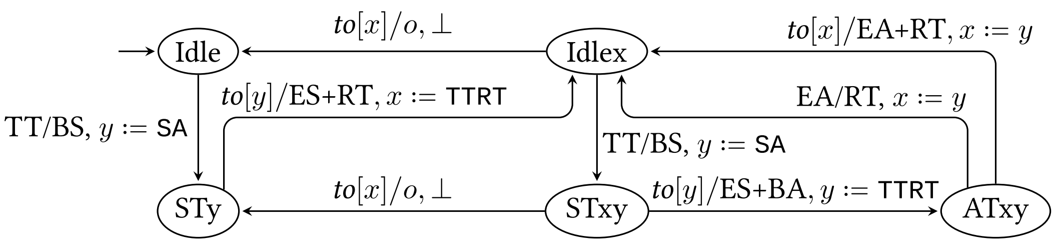

Idle
Idlex
STy
STxy
ATxy
$to[x]/o, \perp$
$to[x]/\text{EA+RT}, x := y$
$to[y]/\text{ES+RT}, x := \texttt{TTRT}$
$\text{EA/RT}, x := y$
$\text{TT/BS}, y := \texttt{SA}$
$\text{TT/BS}, y := \texttt{SA}$
$to[x]/o, \perp$
$to[y]/\text{ES+BA}, y := \texttt{TTRT}$

Figure 10.14: A generalized MMT model of one FDDI station.

We have implemented the $L^{\#}_{\text{MMT}}$ algorithm as an open-source tool.[16] As we do not yet have a timed conformance testing algorithm for checking symbolic equivalence between a hypothesis and the teacher's MMT, we utilize a BFS algorithm to check for equivalence between the two MMTs.[17] We have evaluated the performance of our tool on a selection of both real and synthetic benchmarks. We discuss our selection of benchmarks and metrics, report our results, and then contextualize the same.

16: Available on GitLab at: `https://gitlab.science.ru.nl/bharat/mmt_lsharp` and Zenodo: `https://zenodo.org/records/10647628`.

17: That is, seeking a difference in behavior in the product of the zone MMT of the hypothesis and the zone MMT of the teacher's machine.

10.5.1. Selected benchmarks

We use the AKM, TCP and Train benchmarks from [VBE21], and the CAS, Light and PC benchmarks from [APT20]. These benchmark have also been used for experimental evaluation by [VBE21; Wag23; KKG23] and can be described as Mealy machines with a single timer (MM1Ts, for short). We introduce two additional benchmarks with 2 timers: a model of an FDDI station, and the MMT of Figure 10.1.[18] Finally, we learned instances of the Oven and WSN Mealy machines with *local* timers (MMLTs, for short) benchmarks from [KKG23]. We have modified the timing parameters to generate smaller MM1Ts.

[APT20]: Aichernig et al. (2020), "From Passive to Active: Learning Timed Automata Efficiently"

[VBE21]: Vaandrager et al. (2021), "Learning Mealy Machines with One Timer"

[Wag23]: Waga (2023), "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization"

[KKG23]: Kogel et al. (2023), "Learning Mealy Machines with Local Timers"

18: The zone MMT is computed on the fly when needed.

FDDI protocol

By far the largest benchmark that is learned by Waga [Wag23] is a fragment of the FDDI communication protocol [Joh87], based on a timed automaton model described in [Daw+95]. FDDI (Fiber Distributed Data Interface) is a protocol for a token ring that is composed of N identical stations. Figure 10.14 shows a generalized MMT-translation (*i.e.*, an MMT where timers can be renamed along the transitions; for instance, y becomes x when going from `ATxy` to `Idlex`) of the timed automaton model for a single station from [Daw+95].[19]

[Joh87]: Johnson (1987), "Proof that Timing Requirements of the FDDI Token Ring Protocol are Satisfied"

[Daw+95]: Daws et al. (1995), "The Tool KRONOS"

19: We refer to Section C.13.1 for the definition of generalized MMTs.

In the initial state `Idle`, the station is waiting for the token. When the token arrives (`TT`), the station begins with transmission of synchronous messages (`BS`). A timer y ensures that synchronous transmission ends (`ES`) after exactly `SA` time units, for some constant `SA` (= Synchronous Allocation). The station also maintains a timer x, that expires exactly `TTRT`+`SA` time units after the previous receipt of the token, for some constant `TTRT` (= Target Token Rotation Timer).[20]

20: `SA` and `TTRT` can be seen as parameters for the model.

When synchronous transmission ends and timer x has not expired yet, the station has the possibility to begin transmission of asynchronous messages (`BA`). Asynchronous transmission must end (`EA`) and the token must be returned

Model	$\|Q\|$	$\|I\|$	$\|X\|$	$\|\mathbf{WQ^s}\|$	$\|\mathbf{OQ^s}\|$	$\|\mathbf{EQ^s}\|$	Time[msecs]	$\|\mathbf{MQ}\|$ [Wag23]	$\|\mathbf{EQ}\|$ [Wag23]
AKM	4	5	1	22	35	2	684	12263	11
CAS	8	4	1	60	89	3	1344	66067	17
Light	4	2	1	10	13	2	302	3057	7
PC	8	9	1	75	183	4	2696	245134	23
TCP	11	8	1	123	366	8	3182	11300	15
Train	6	3	1	32	28	3	1559		
MMT of Figure 10.1	3	1	2	11	5	2	1039	-	-
FDDI 1-station	9	2	2	32	20	1	1105	118193	8
Oven	12	5	1	907	317	3	9452	-	-
WSN	9	4	1	175	108	4	3291	-	-

Table 10.1: Experimental Results. The columns give the considered model, followed by its numbers of states, input symbols, and active timers, as well as the number of queries and the total time needed to learn a machine. Finally, the last three columns give the number of queries needed for Waga's approach.

(RT) at the latest when x expires. In location ATxy, timer x will expire before timer y (we may formally prove this by computing the zone MMT):

1. The value of y in location ATxy is at most TTRT.
2. Hence, the value of x in location Idlex is at most TTRT.
3. So x is at most TTRT upon arrival in location STxy, and at most TTRT—SA upon arrival in location ATxy.
4. Thus x is smaller than y in location ATxy and will expire first.

Upon entering location Idlex, we ensure that timer x will expire exactly TTRT+SA time units after the previous TT event. In a FDDI token ring of size N, an RT event of station i will instantly trigger a TT event of station $(i+1)$ mod N. In [Wag23], the instance with two stations, SA $= 20$, and TTRT $= 100$ was considered. We did not include the FDDI two process benchmark from [Wag23] as its equivalent MMT may (re)start two timers in the same transition, leaving it out of scope of our setting.

10.5.2. Metrics

For each experiment, we record the number of $\mathbf{OQ^s}$, $\mathbf{WQ^s}$, $\mathbf{EQ^s}$, and the time taken to finish the experiment. Note that, in practice, a $\mathbf{WQ^s}$, in addition to returning the list of timeouts and their constraints, also provides the outputs of the timeout transitions. This is straightforward, as a $\mathbf{WQ^s}$ must necessarily trigger the timeouts in order to observe them. Thus, we do not count the $\mathbf{OQ^s}$ associated with a $\mathbf{WQ^s}$.

10.5.3. Results

Table 10.1 lists the results of our experiments, and also the number of concrete membership and equivalence queries used by Waga's [Wag23]. Comparison of learning algorithms for timed systems is complicated. First of all, we need to convert the numbers of symbolic $L^{\#}_{\text{MMT}}$ queries to concrete queries. This can be done using the bounds given in Section C.9. For instance, one can compute that we need at most 1282 concrete queries to do our symbolic wait and output queries in total, for the FDDI protocol.[21] Observe that, for MM1Ts, each symbolic query can be implemented using a single concrete query (see [VBE21, Lemma 3]).

21: This can be obtained from the bounds of Lemmas C.9.1 to C.9.3.

[VBE21]: Vaandrager et al. (2021), "Learning Mealy Machines with One Timer"

Several algorithms presented in the literature learn TAs [Wag23; APT20; XAZ22; An+20]. Typically, a TA model of some system will have different numbers of states and transitions than an MMT model. In general, Mealy machines tend to be more compact than automata. In our timed case, however, the use of timers may required more states compared to using clocks, in order to correctly encode the timed function. Therefore we cannot just compare numbers of queries.

[Wag23]: Waga (2023), "Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization"

[APT20]: Aichernig et al. (2020), "From Passive to Active: Learning Timed Automata Efficiently"

[XAZ22]: Xu et al. (2022), "Active Learning of One-Clock Timed Automata Using Constraint Solving"

[An+20]: An et al. (2020), "Learning One-Clock Timed Automata"

As a final complication, observe that equivalence queries can be implemented in different ways, which may affect the total number of queries required for learning. MMLTs [KKG23] can be converted to equivalent MM1Ts [VBE21], but this may blow up of the number of states. Since $L^{\#}_{\text{MMT}}$ learns the MM1Ts, it is less efficient than the MMLT learner of [KKG23] which learns the more compact MMLT representations. However, $L^{\#}_{\text{MMT}}$ can handle a larger class of models.

[KKG23]: Kogel et al. (2023), "Learning Mealy Machines with Local Timers"

[VBE21]: Vaandrager et al. (2021), "Learning Mealy Machines with One Timer"

10.6. Conclusion

In this chapter, we presented an active learning algorithm for Mealy machines with timers, using symbolic queries. This algorithm infers a finite tree encoding the runs of the teacher's MMT $\mathcal{M}$ and from which an MMT $\mathcal{H}$ can be constructed such that $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{H}$. By Proposition 9.4.5, $\mathcal{M}$ and $\mathcal{H}$ are thus timed equivalent. The learner requires a number of queries that is polynomial in the number of states of $\mathcal{M}$, in the number of input symbols, and in the length of the longest counterexample returned for an equivalence query, and factorial in the number of timers of $\mathcal{M}$.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

There are many possibilities for future work. One could try to extend the framework to MMTs where multiple timers can be started on a single transition, or to generalized MMTs where timers can be renamed. Without changing the target model, there are still some improvements to be done, such as constructing an MMT from the observation tree in every case without having to use generalized MMTs. The way counterexamples are processed may be improved as well, by trying to apply a binary search, instead of a linear search as we presented here. Finally, applying the learning algorithm on more complex examples (necessitating many timers) may yield interesting results. In particular, one could compare the efficiency of model checking algorithm on the resulting MMTs against timed automata.

Technical details and proofs of Chapters 9 and 10 C.

This chapter, based on [Bru+23; Bru+24], contains the technical details and proofs that were not given in Chapters 9 and 10. That is, it serves as the appendix of the previous two chapters.

[Bru+23]: Bruyère et al. (2023), "Automata with Timers"
[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

Chapter contents

C.1. From symbolic words to concrete runs

Let $\mathtt{w} = \mathtt{i}_1 \dots \mathtt{i}_n$ be a symbolic word over $\mathtt{A}$. Let us explain how to convert $\mathtt{w}$ into a run $q_0 \xrightarrow{w}$ using concrete timeout symbols such that $\overline{w} = \mathtt{w}$, if such a run exists in $\mathcal{M}$. Assume that we were able to process $\mathtt{i}_1 \cdots \mathtt{i}_k$ with $k \in \{0, \dots, n-1\}$

and reach the state q_k such that $q_0 \xrightarrow{i_1 \dots i_k} q_k \in \mathit{runs}(\mathcal{M})$ and $\overline{i_1 \dots i_k} = \mathtt{i_1} \cdots \mathtt{i_k}$, and that we want to convert $\mathtt{i_{k+1}}$.

- If $\mathtt{i_{k+1}}$ is an input, we can simply take the transition $q_k \xrightarrow{i_{k+1}} q_{k+1}$ of the complete MMT with $i_{k+1} = \mathtt{i_{k+1}}$.
- Otherwise, $\mathtt{i_{k+1}} = \mathit{to}[j]$ for some $j \in \mathbb{N}^{>0}$.
  - Suppose first that $j \leq k$. We actually want to read the timeout of the timer (re)started on the transition from q_{j-1} to q_j. Let u be its update. On the one hand, if $u = (x, c)$ and $q_{j-1} \xrightarrow{i_j \cdots i_k \cdot \mathit{to}[x]}$ is x-spanning, then we actually read $\mathit{to}[x]$, *i. e.*, we take the transition $q_k \xrightarrow{i_{k+1}} q_{k+1}$ with $i_{k+1} = \mathit{to}[x]$. On the other hand, if u is $\bot$ or the sub-run from q_{j-1} is not x-spanning, then the symbol $\mathtt{i_{k+1}}$ of $\mathtt{w}$ does not make sense and therefore there exists no run $q_0 \xrightarrow{w}$ such that $\overline{w} = \mathtt{w}$.
  - Suppose now that $j > k$. Again, the symbol $\mathtt{i_{k+1}}$ does not make sense and there exists no run $q_0 \xrightarrow{w}$ such that $\overline{w} = \mathtt{w}$.

We repeat this process in a way to obtain a run $q_0 \xrightarrow{w} q_n \in \mathit{runs}(\mathcal{M})$ such that $\overline{w} = \mathtt{w}$ or concluding that such a run does not exist. In the first case, in an abuse of notation, we write $q_0 \xrightarrow{\mathtt{w}} q_n$ for the run reading the symbolic word $\mathtt{w}$, and we say that it is *feasible* whenever $q_0 \xrightarrow{w} q_n$ is feasible. We also say that the sub-run $q_{j-1} \xrightarrow{\mathtt{i_j} \cdots \mathtt{i_k} \cdot \mathit{to}[j]} q_{k+1}$ is *spanning* whenever $q_{j-1} \xrightarrow{i_j \cdots i_k \cdot \mathit{to}[x]} q_{k+1}$ is x-spanning.

C.2. Proof of Proposition 9.4.5

We show Proposition 9.4.5, *i. e.*, that symbolic equivalence implies timed equivalence. Moreover, we give a counterexample for the other direction. That is, we prove that timed equivalence does not imply symbolic equivalence.

Proposition 9.4.5. *Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.*

Proof. Towards a contradiction, assume $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$ but $\mathcal{M} \overset{\text{time}}{\not\approx} \mathcal{N}$. Then, there must exist a tiw w such that $\mathit{toutputs}^{\mathcal{M}}(w) \neq \mathit{toutputs}^{\mathcal{N}}(w)$. Without loss of generality, assume there is a timed run

$$\rho = (q_0^{\mathcal{M}}, \emptyset) \xrightarrow{d_1} (q_0^{\mathcal{M}}, \emptyset) \xrightarrow[u_1]{i_1/o_1} \cdots \xrightarrow[u_n]{i_n/o_n} (q_n, \kappa_n) \\ \xrightarrow{d_{n+1}} (q_n, \kappa_n - d_{n+1}) \in \mathit{tiwruns}^{\mathcal{M}}(w)$$

such that $\mathit{tow}(\rho) = d_1 \cdot o_1 \cdots d_n \cdot o_n \cdot d_{n+1}$ and $\mathit{tow}(\rho) \notin \mathit{toutputs}^{\mathcal{N}}(w)$.[1] Let $\mathtt{w} = \mathtt{i_1} \cdots \mathtt{i_n}$ be the symbolic word of $i_1 \cdots i_n$.

1: Recall that $\mathit{tow}(\rho)$ denotes the tow produced by the timed run ρ.

Let us consider the longest possible timed run of $\mathcal{N}$

$$\rho' = (q_0^{\mathcal{N}}, \emptyset) \xrightarrow{d_1} (q_0^{\mathcal{N}}, \emptyset) \xrightarrow[u'_1]{i'_1/o_1} \cdots \xrightarrow[u'_j]{i'_j/o_j} (q'_j, \kappa'_j) \xrightarrow{d_{j+1}} (q'_j, \kappa'_j - d_{j+1})$$

such that

- it reads a prefix of $i_1 \cdots i_n$ (up to the names of the timers) with

  $$i'_k = \begin{cases} i_k & \text{if } i_k \in I \\ to[x'] & \text{for some } x' \in \chi^{\mathcal{N}}(q'_{k-1}) \text{ if } i_k = to[x] \text{ with } x \in \chi^{\mathcal{M}}(q_{k-1}) \end{cases}$$

  for all $k \in \{1, \dots, j\}$,
- the delays d_k, $k \in \{1, \dots, j+1\}$, and the outputs o_k, $k \in \{1, \dots, j\}$, are the same as in the timed run ρ, and
- the symbolic word of $i'_1 \cdots i'_j$ is equal to $\mathtt{i}_1 \cdots \mathtt{i}_j$.

Such a timed run ρ' exists with $0 \leq j < n$. Towards a contradiction, let us show that we can extend it.
First, we argue that for any $d \in \mathbb{R}^{\geq 0}$

$$\begin{aligned} & \exists x \in \chi^{\mathcal{M}}(q_j) : (\kappa_j - d)(x) = 0 \\ \Leftrightarrow \quad & \exists x' \in \chi^{\mathcal{N}}(q'_j) : (\kappa'_j - d)(x') = 0. \end{aligned} \tag{C.2.i}$$

We show the $\Rightarrow$ direction. The other direction can be obtained with similar arguments. Since $(\kappa_j - d)(x) = 0$, we have that $x \in \chi_0^{\mathcal{M}}(q_j)$. As $\mathcal{M}$ is complete, it follows that $q_j \xrightarrow{to[x]} \in \mathit{runs}(\mathcal{M})$ and we can take the transition $(q_j, \kappa_j - d) \xrightarrow{to[x]}$. Thus, for some $k \in \{1, \dots, j-1\}$, the sub-run

$$q_{k-1} \xrightarrow[(x,c)]{i_k} \cdots \xrightarrow{i_j} q_j \xrightarrow{to[x]}$$

is x-spanning. Let $\mathtt{i}_1 \cdots \mathtt{i}_j \cdot \mathtt{i}$ be the symbolic word of $i_1 \cdots i_j \cdot to[x]$. As $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, we deduce that there exists some timer x' such that the run $q_0^{\mathcal{N}} \xrightarrow{i'_1 \cdots i'_j \cdot to[x']}$ is feasible with

$$\overline{i'_1 \cdots i'_j \cdot to[x']} = \mathtt{i}_1 \cdots \mathtt{i}_j \cdot \mathtt{i},$$

x' is enabled in q'_j, and the sub-run

$$q'_{k-1} \xrightarrow[(x',c')]{i'_k} \cdots \xrightarrow{i'_j} q'_j \xrightarrow{to[x']}$$

is x'-spanning with $c' = c$. Since the delays in the timed run ρ' are the same as in ρ and x, x' are both started at the same constant c along the j-th transition, it naturally follows that $(\kappa'_j - d)(x') = 0$. That is, (C.2.i) holds.
Let us now move towards the actual contradiction: we argue that we can replicate the action i_{j+1} and the subsequent delay at the end of the timed run of $\mathcal{N}$, *i.e.*, ρ' is not the longest possible run described above. Consider thus the action i_{j+1}. We have two cases:

- If $i_{j+1} \in I$, the transition $(q'_j, \kappa'_j - d_{j+1}) \xrightarrow{i'_{j+1}} (q'_{j+1}, \kappa'_{j+1})$ with $i'_{j+1} =$

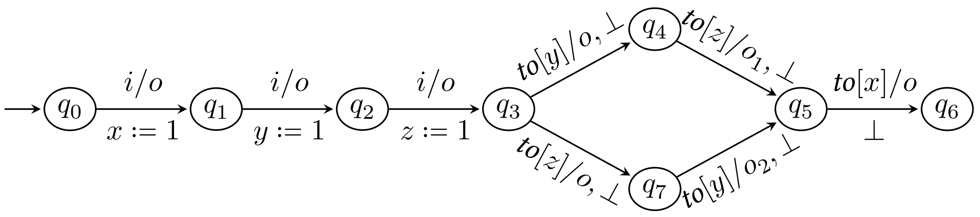


Figure C.1: An MMT with $\chi(q_0) = \chi(q_6) = \emptyset$, $\chi(q_1) = \chi(q_5) = \{x\}$, $\chi(q_2) = \chi(q_7) = \{x, y\}$, $\chi(q_3) = \{x, y, z\}$, $\chi(q_4) = \{x, z\}$. Every missing transition $q \xrightarrow[u]{i/\omega} p$ to obtain a complete MMT is such that $p = q_6, \omega = o$, and $u = \perp$.

i_{j+1} is defined as $\mathcal{N}$ is complete by hypothesis.

- If $i_{j+1} = to[x]$ for some $x \in \chi_0^{\mathcal{M}}(q_j)$, we have by (C.2.i) that there exists $x' \in \chi_0^{\mathcal{N}}(q'_j)$ such that $(\kappa'_j - d_{j+1})(x') = 0$. As $\mathcal{N}$ is complete, we can thus take the transition

$$(q'_j, \kappa'_j - d_{j+1}) \xrightarrow{to[x']} (q'_{j+1}, \kappa'_{j+1}).$$

By the previous arguments establishing (C.2.i), it follows that $\overline{i'_1 \cdots i'_{j+1}} = \mathtt{i}_1 \cdots \mathtt{i}_{j+1}$. As $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, we get that the output o' of $q'_j \xrightarrow{i'_{j+1}/o'} q'_{j+1}$ is equal to o_{j+1}.

It remains to prove that the delay transition $(q'_{j+1}, \kappa'_{j+1}) \xrightarrow{d_{j+2}}$ is possible. Assume the contrary, *i.e.*, there exists a timer $x' \in \chi^{\mathcal{N}}(q'_{j+1})$ such that $\kappa'_{j+1}(x') < d_{j+2}$. Let $d' = \kappa'_{j+1}(x')$. We thus have that $(\kappa'_{j+1} - d')(x') = 0$. By (C.2.i) applied to q_{j+1} and q'_{j+1}, there must exist a timer x such that $(\kappa_{j+1} - d')(x) = 0$, *i.e.*, it is not possible to wait d_{j+2} units of time in (q_{j+1}, κ_{j+1}) and we have a contradiction.

We are thus able to extend the timed run ρ' which leads to the contradiction. We conclude that the symbolic equivalence implies the timed equivalence. □

C.2.1. Counterexample for timed equivalence implies symbolic equivalence

Let $\mathcal{M}$ be the MMT of Figure C.1. We make $\mathcal{M}$ complete by adding $q \xrightarrow[\perp]{i/o} q_6$ for every missing transition. Observe that $O = \{o, o_1, o_2\}$ and that $q_4 \xrightarrow{to[z]}$ outputs o_1 while $q_7 \xrightarrow{to[y]}$ outputs o_2. Moreover, let $\mathcal{N}$ be a copy of $\mathcal{M}$ such that o_1 and o_2 are swapped. Let us argue that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ but $\mathcal{M} \overset{\text{sym}}{\not\approx} \mathcal{N}$, starting with the latter. We write $q_j^{\mathcal{M}}$ and $q_j^{\mathcal{N}}$ to distinguish the states of $\mathcal{M}$ and $\mathcal{N}$.

Let $\mathtt{w} = i \cdot i \cdot i \cdot to[2] \cdot to[3]$ be a symbolic word, inducing the following runs:

$$q_0^{\mathcal{M}} \xrightarrow[(x,1)]{i/o} q_1^{\mathcal{M}} \xrightarrow[(y,1)]{i/o} q_2^{\mathcal{M}} \xrightarrow[(z,1)]{i/o} q_3^{\mathcal{M}} \xrightarrow[\perp]{to[y]/o} q_4^{\mathcal{M}} \xrightarrow[\perp]{to[z]/o_1} q_5^{\mathcal{M}}$$
$$q_0^{\mathcal{N}} \xrightarrow[(x,1)]{i/o} q_1^{\mathcal{N}} \xrightarrow[(y,1)]{i/o} q_2^{\mathcal{N}} \xrightarrow[(z,1)]{i/o} q_3^{\mathcal{N}} \xrightarrow[\perp]{to[y]/o} q_4^{\mathcal{N}} \xrightarrow[\perp]{to[z]/o_2} q_5^{\mathcal{N}}.$$

Hence, $\mathcal{M} \overset{\text{sym}}{\not\approx} \mathcal{N}$ as the last pair of transitions has different outputs.

So, it remains to show that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$. Clearly, any tiw induces the same run in both $\mathcal{M}$ and $\mathcal{N}$ (up to the outputs), as they have exactly the same transitions. That is, any tiw w is such that $q_0^{\mathcal{M}} \xrightarrow{w} q_j^{\mathcal{M}}$ if and only if $q_0^{\mathcal{N}} \xrightarrow{w} q_j^{\mathcal{N}}$. Moreover, outputs are the same in $\mathcal{M}$ and $\mathcal{N}$, except that o_1 and o_2 are swapped. So, let us focus on the transitions $q_4^{\mathcal{M}} \xrightarrow{\mathit{to}[z]/o_1}$ and $q_7^{\mathcal{M}} \xrightarrow{\mathit{to}[y]/o_2}$. It is not hard to see that the only way these transitions are triggered is to start all timers x, y, and z without any delay in between, to go through in either $(q_4^{\mathcal{M}}, x = 0, z = 0)$ or $(q_7^{\mathcal{M}}, x = 0, y = 0)$, and to trigger the $\mathit{to}[z]$ and $\mathit{to}[y]$ transitions, respectively, *i.e.*, we must take the following two timed runs in $\mathcal{M}$:

$$\begin{aligned}
(q_0^{\mathcal{M}}, \emptyset) &\xrightarrow{0} (q_0^{\mathcal{M}}, \emptyset) \xrightarrow[(x,1)]{i/o} (q_1^{\mathcal{M}}, x = 1) \xrightarrow{0} (q_1^{\mathcal{M}}, x = 1)\\
&\xrightarrow[(y,1)]{i/o} (q_2^{\mathcal{M}}, x = 1, y = 1) \xrightarrow{0} (q_2^{\mathcal{M}}, x = 1, y = 1)\\
&\xrightarrow[(z,1)]{i/o} (q_3^{\mathcal{M}}, x = 1, y = 1, z = 1)\\
&\xrightarrow{1} (q_3^{\mathcal{M}}, x = 0, y = 0, z = 0)\\
&\xrightarrow[\perp]{\mathit{to}[y]/o} (q_4^{\mathcal{M}}, x = 0, z = 0) \xrightarrow{0} (q_4^{\mathcal{M}}, x = 0, z = 0)\\
&\xrightarrow[\perp]{\mathit{to}[z]/o_1} (q_5^{\mathcal{M}}, x = 0) \xrightarrow{0} (q_5^{\mathcal{M}}, x = 0) \in \mathit{truns}(\mathcal{M})
\end{aligned}$$

and

$$\begin{aligned}
(q_0^{\mathcal{M}}, \emptyset) &\xrightarrow{0} (q_0^{\mathcal{M}}, \emptyset) \xrightarrow[(x,1)]{i/o} (q_1^{\mathcal{M}}, x = 1) \xrightarrow{0} (q_1^{\mathcal{M}}, x = 1)\\
&\xrightarrow[(y,1)]{i/o} (q_2^{\mathcal{M}}, x = 1, y = 1) \xrightarrow{0} (q_2^{\mathcal{M}}, x = 1, y = 1)\\
&\xrightarrow[(z,1)]{i/o} (q_3^{\mathcal{M}}, x = 1, y = 1, z = 1)\\
&\xrightarrow{1} (q_3^{\mathcal{M}}, x = 0, y = 0, z = 0)\\
&\xrightarrow[\perp]{\mathit{to}[z]/o} (q_7^{\mathcal{M}}, x = 0, y = 0) \xrightarrow{0} (q_7^{\mathcal{M}}, x = 0, y = 0)\\
&\xrightarrow[\perp]{\mathit{to}[y]/o_2} (q_5^{\mathcal{M}}, x = 0) \xrightarrow{0} (q_5^{\mathcal{M}}, x = 0) \in \mathit{truns}(\mathcal{M}).
\end{aligned}$$

We can then obtain similar runs in $\mathcal{N}$, up to a swap of o_1 and o_2:

$$\begin{aligned}
(q_0^{\mathcal{N}}, \emptyset) &\xrightarrow{0} (q_0^{\mathcal{N}}, \emptyset) \xrightarrow[(x,1)]{i/o} (q_1^{\mathcal{N}}, x = 1) \xrightarrow{0} (q_1^{\mathcal{N}}, x = 1)\\
&\xrightarrow[(y,1)]{i/o} (q_2^{\mathcal{N}}, x = 1, y = 1) \xrightarrow{0} (q_2^{\mathcal{N}}, x = 1, y = 1)\\
&\xrightarrow[(z,1)]{i/o} (q_3^{\mathcal{N}}, x = 1, y = 1, z = 1)\\
&\xrightarrow{1} (q_3^{\mathcal{N}}, x = 0, y = 0, z = 0)\\
&\xrightarrow[\perp]{\mathit{to}[y]/o} (q_4^{\mathcal{N}}, x = 0, z = 0) \xrightarrow{0} (q_4^{\mathcal{N}}, x = 0, z = 0)\\
&\xrightarrow[\perp]{\mathit{to}[z]/o_2} (q_5^{\mathcal{N}}, x = 0) \xrightarrow{0} (q_5^{\mathcal{N}}, x = 0) \in \mathit{truns}(\mathcal{N})
\end{aligned}$$

and

$$\begin{aligned}
(q_0^{\mathcal{N}}, \emptyset) &\xrightarrow{0} (q_0^{\mathcal{N}}, \emptyset) \xrightarrow[(x,1)]{i/o} (q_1^{\mathcal{N}}, x = 1) \xrightarrow{0} (q_1^{\mathcal{N}}, x = 1) \\
&\xrightarrow[(y,1)]{i/o} (q_2^{\mathcal{N}}, x = 1, y = 1) \xrightarrow{0} (q_2^{\mathcal{N}}, x = 1, y = 1) \\
&\xrightarrow[(z,1)]{i/o} (q_3^{\mathcal{N}}, x = 1, y = 1, z = 1) \\
&\xrightarrow{1} (q_3^{\mathcal{N}}, x = 0, y = 0, z = 0) \\
&\xrightarrow[\bot]{\mathit{to}[z]/o} (q_7^{\mathcal{N}}, x = 0, y = 0) \xrightarrow{0} (q_7^{\mathcal{N}}, x = 0, y = 0) \\
&\xrightarrow[\bot]{\mathit{to}[y]/o_1} (q_5^{\mathcal{N}}, x = 0) \xrightarrow{0} (q_5^{\mathcal{N}}, x = 0) \in \mathit{truns}(\mathcal{N}).
\end{aligned}$$

Hence, any tiw w inducing the first run in $\mathcal{M}$ necessarily induces the second run, too (and the runs triggering $\mathit{to}[x]$). Moreover, w also induces the two runs in $\mathcal{N}$ (and the runs triggering $\mathit{to}[x]$). We thus conclude that $\mathit{toutputs}^{\mathcal{M}}(w) = \mathit{toutputs}^{\mathcal{N}}(w)$ for every tiw w. That is, $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ and $\mathcal{M} \overset{\text{sym}}{\not\approx} \mathcal{N}$.

To conclude, we highlight that we needed to consider null delays in the timed runs.[2] It is unknown whether $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ implies $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$ when we can assume that all delays are positive.

2: That is, $\mathcal{M}$ is not race-avoiding, as is defined in Section 9.6.

C.3. Proof of PSPACE lower bound of Theorem 9.5.1

Theorem 9.5.1. *The reachability problem for MMTs is* PSPACE*-complete.*

A *LBTM* $\mathcal{A} = (\Sigma, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, F^{\mathcal{A}}, T)$ is a nondeterministic Turing machine which can only use $|w|$ cells of the tape to determine whether an input w is accepted. Formally, Σ is a finite alphabet, $Q^{\mathcal{A}}$ is a finite set of states, $q_0^{\mathcal{A}}$ and $F^{\mathcal{A}}$ are the initial and final states respectively, and

$$T \subseteq (Q^{\mathcal{A}} \times \Sigma) \times (\Sigma \times \{L, R\} \times Q^{\mathcal{A}})$$

is the transition relation. A configuration of $\mathcal{A}$ is a triple $(q, w, i) \in Q^{\mathcal{A}} \times \Sigma^* \times \mathbb{N}^{>0}$ where q denotes the current control state, $w = w_1 \dots w_n$ is the content of the tape, and i is the position of the tape head. We say a transition $(q, \alpha, \alpha', D, q') \in T$ is enabled in a configuration (q, w, i) if $w_i = \alpha$. In that case, taking the transition results in the machine reaching the new configuration (q', w', i') with

- $w'_i = \alpha'$,
- $w'_j = w_j$ for all $j \neq i$, and
- $i' = i + 1$ if $D = R$ or $i' = i - 1$ otherwise.

A given word w is said to be accepted by $\mathcal{A}$ if there is a sequence of transitions from $(q_0^{\mathcal{A}}, w, 1)$ to a configuration of the form $(F^{\mathcal{A}}, w', i)$. Deciding whether a given LBTM accepts a given word is PSPACE-complete [HU79].

[HU79]: Hopcroft et al. (1979), *Introduction to Automata Theory, Languages and Computation*

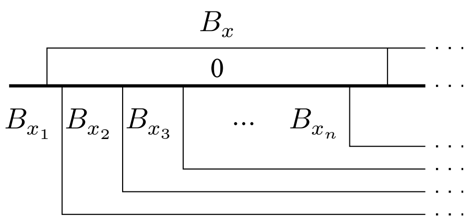


Figure C.2: Starting blocks.

Lower bound of Theorem 9.5.1. We show that the acceptance problem for LBTMs can be reduced in polynomial time to the reachability problem for MMTs. The proof is inspired by the one presented by Aceto and Laroussinie for the fact that reachability is PSPACE-hard for timed automata [AL02, Section 3.1].

[AL02]: Aceto et al. (2002), "Is your model checker on time? On the complexity of model checking for timed modal logics"

Let $\mathcal{A} = (\Sigma, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, F^{\mathcal{A}}, T)$ be an LBTM and let $w \in \Sigma^*$ be an input word with $|w| = n$. From $\mathcal{A}$ and w we are going to build, in polynomial time, an MMT $\mathcal{M}_{\mathcal{A},w}$ such that w is accepted by $\mathcal{A}$ if and only if there exists a timed run that ends in a specific state r_{done} of $\mathcal{M}_{\mathcal{A},w}$.[3]

3: In the following, we rely on the notion of blocks to pass on the intuition.

Let $\Sigma = \{a_1, \dots, a_k\}$. Then, every $2k + 2$ time units the MMT will simulate a single step of the LBTM. This is what we call a *phase*.

Let $T = \{t_1, \dots, t_m\}$ be the set of transitions of $\mathcal{A}$. Then, $\mathcal{M}_{\mathcal{A},w}$ has inputs $I = \{go\} \cup T$ and timers $X = \{x, x_1, \dots, x_n\}$. Now, a state of $\mathcal{M}_{\mathcal{A},w}$ is an element of $\{r_0, \dots, r_n, r_{done}, r_{sink}\}$ (where r_0 is the initial state) or a tuple $\langle q, i, symbol, clock\rangle$, where:

- $q \in Q^{\mathcal{A}}$ records the current state of the LBTM,
- $i \in \{1, \dots, n\}$ records the current position of the tape head,
- $symbol \in \{0, \dots, k\}$ records the index of the last symbol read from the tape (0 when no symbol has been read or the last read symbol has been processed), and
- $clock \in \{0, 1, \dots, 2k + 1\}$.

The MMT starts with an initialization phase in which it goes through states r_0 to r_n to start all the timers via a sequence of go-inputs. Execution begins with a go-input that starts timer x: $r_0 \xrightarrow[(x,1)]{go} r_1$. In order to reach state r_{done}, all the other timers need to be started before timer x times out. We use timer x_i, for $1 \leq i \leq n$, to record the value of the i-th tape cell: if this value is symbol $a_j \in \Sigma$, then timer x_i will time out when $\lfloor clock/2 \rfloor = j$ (it will become clear later how this is possible). Timer x_i is started in state r_i and set to its appropriate value:

- if $i < n$, we have the transition $r_i \xrightarrow[(x_i,2j)]{go} r_{i+1}$ that initializes timer x_i with value $2j$,
- if $i = n$, we have the transition $r_n \xrightarrow[(x_n,2j)]{go} \langle q_0^{\mathcal{A}}, 1, 0, 0\rangle$ that initializes timer x_n with the same value $2j$ and starts the computation of the LBTM.

All timeout transitions from r_i, $1 \leq i \leq n$, go to r_{sink}. This ensures that — in order to reach state r_{done}— all timers x_i are initialized. Hence, all timed

runs in $\mathcal{M}_{\mathcal{A},w}$ that reach r_{done} have the same starting blocks, as depicted in Figure C.2: an x-block B_x and n x_i-blocks B_{x_i}.
We use timer x to advance the value of *clock* that runs cyclically from 0 to $2k+1$: $\mathit{clock} > 0$ or $\mathit{symbol} = 0$ implies that

$$\langle q, i, \mathit{symbol}, \mathit{clock}\rangle \xrightarrow[(x,1)]{\mathit{to}[x]} \langle q, i, \mathit{symbol}, (\mathit{clock}+1) \bmod 2k+2\rangle. \tag{C.3.i}$$

(It will become clear later why the condition on this transition is required.)
When timer x_ℓ times out, for some $\ell \neq i$, then we just restart it so that it will times out at exactly the same point in the next phase:

$$\langle q, i, \mathit{symbol}, \mathit{clock}\rangle \xrightarrow[(x_\ell,2k+2)]{\mathit{to}[x_\ell]} \langle q, i, \mathit{symbol}, \mathit{clock}\rangle.$$

When timer x_i times out, then we restart it in the same way, but in addition we store the index of the symbol that it encodes in the state of the MMT:

$$\langle q, i, \mathit{symbol}, \mathit{clock}\rangle \xrightarrow[(x_i,2k+2)]{\mathit{to}[x_i]} \langle q, i, \lfloor \mathit{clock}/2\rfloor, \mathit{clock}\rangle.$$

In order to see why this is true, suppose timer x_i has been started at time d with value $2j$. Then, $d \in [0,1]$ (see Figure C.2) and x_i will expire at time $d' = d + 2j$, so $d' \in [2j, 2j+1]$. At this time, the value of *clock* will be either $2j$ or $2j+1$, and thus $j = \lfloor \mathit{clock}/2\rfloor$.
When the value of *clock* becomes 0 again (a next phase begins), the LBTM $\mathcal{M}$ has read a symbol from the tape, so $\mathit{symbol} > 0$, and $\mathcal{M}$ may (nondeterministically) take a transition. For each transition $t = (q, \alpha, \alpha', q', L)$ of $\mathcal{M}$, with $\alpha = a_{\mathit{symbol}}$ and $\alpha' = a_j$, the MMT has a transition:

$$\langle q, i, \mathit{symbol}, 0\rangle \xrightarrow[(x_i,2j)]{t} \langle q', i-1, 0, 0\rangle.$$

The MMT also has transitions, mutatis mutandis, for each $t = (q, \alpha, \alpha', q', R)$ of the LBTM. In order to ensure that these transitions are taken before timer x times out, we add transitions:

$$\mathit{symbol} > 0 \Rightarrow \langle q, i, \mathit{symbol}, 0\rangle \xrightarrow{\mathit{to}[x]} r_{\mathit{sink}}. \tag{C.3.ii}$$

The condition $\mathit{clock} > 0$ or $\mathit{symbol} = 0$ before (C.3.i) and the condition $\mathit{symbol} > 0$ from (C.3.ii) both ensure that we see first action t and then action $\mathit{to}[x]$ in a timed run that reaches r_{done}.
In Figure C.3, we fix $k = 2$, and, for a timed run ρ that reaches r_{done}, we indicate the sequence of phases, with the cyclic value of *clock* from 0 to $2k+1$. We also indicate the block B_x such that timer x is restarted each time it times out along ρ. Finally, we indicate two x_i-blocks, $B^1_{x_i}$ and $B^2_{x_i}$, such that in $B^1_{x_i}$, timer x_i is started in phase 1, restarted during this phase, and restarted again during phase 2, until it is discarded when $\mathit{clock} = 0$ in phase 3; and in $B^2_{x_i}$, timer x_i is started with a new value dictated by the processed transition of the LBTM. Note that there may be other blocks (for timers x_j, with $j \neq i$) which are not represented in the figure.
As soon as the LBTM reaches $F^{\mathcal{A}}$, the MMT may proceed to its final state

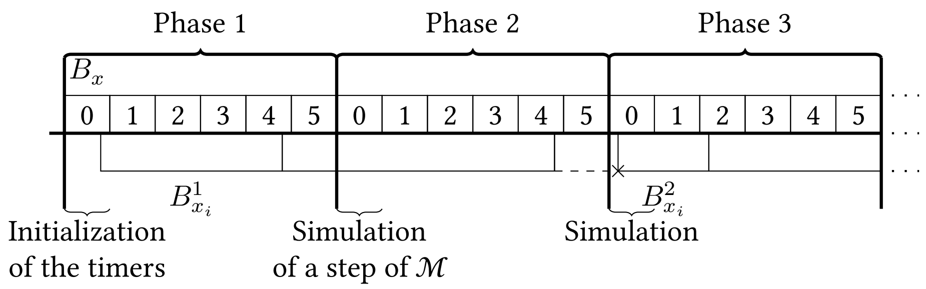


Figure C.3: The beginning of a timed run that reaches the target state with $k = 2$.

r_{done}:

$$\langle q_F, i, symbol, clock\rangle \xrightarrow{go} r_{done}.$$

For all states of the MMT, outgoing transitions for actions that have not been specified lead to r_{sink}.
Finally, we define the active timers in each state of the MMT as follows:

$$\chi(r_0) = \chi(r_{sink}) = \chi(r_{done}) = \emptyset,$$
$$\forall 1 \leq i \leq n : \chi(r_i) = \{x, x_1, x_2, \dots, x_{i-1}\},$$

and

$$\chi(r) = \{x, x_1, x_2, \dots, x_n\}$$

for all the other states r.
It is clear that $\mathcal{M}_{\mathcal{A},w}$ can be constructed from $\mathcal{A}$ and w in polynomial time and that r_{done} is reachable in the MMT if and only if the LBTM accepts w. □

C.4. Proof of Proposition 9.6.9

Proposition 9.6.9. *Let $\mathcal{M}$ be an MMT and $\rho \in ptruns(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.*

Before establishing this result, we prove the following intermediate result.

Lemma C.4.1. *Let G_ρ be the block graph of $\rho \in ptruns(\mathcal{A})$ and B be a block in this graph. It is impossible to wiggle B if and only if B has at least one predecessor and at least one successor in G_ρ.*

Proof. We prove the lemma by showing both directions.

Having a successor and a predecessor implies unwigglable. Suppose that B has a block B' as predecessor and B'' as successor, that is, $B' \prec B$ and $B \prec B''$.[4] Observe that $B \neq B'$ and $B \neq B''$ by the definition of races (see Definition 9.6.1). Let us prove that it is impossible to wiggle B by arguing that it is not feasible to move B to the right nor to the left.
Given $B' \prec B$, let us prove that we cannot move B to the left. We have two cases for the actions $i \in B$ and $i' \in B'$ that participate in the race:

- Action i' occurs before action i along ρ and the sum of the delays

Definition 9.6.1. Let B, B' be two blocks of a padded timed run ρ with timer fates γ and γ'. We say that B and B' *participate in a race* if:

- either there exist actions $i \in B$ and $i' \in B'$ such that the sum of the delays between i and i' in ρ is equal to zero, *i. e.*, no time elapses between them,
- or there exists an action $i \in B$ that is the first action along ρ to discard the timer started by the last action $i' \in B'$ and $\gamma' = ●$, *i. e.*, the timer of B' (re)started by i' reaches value zero when i discards it.

We also say that the actions i and i' participate in this race.

between these two actions is zero. Thus, the delay d before i in ρ is equal to zero and it is impossible to have $d + \epsilon \geq 0$ for any $\epsilon < 0$. This implies that no movement of B to the left is possible.
- The timer x of B (re)started by i reaches value zero when i' discards it. By moving B to the left by some $\epsilon < 0$, x times out and therefore an action *to*$[x]$ occurs, while it does not occur in ρ (as i' discards x). As we want to keep the same untimed trace as for ρ, it is impossible to move B to the left.

With symmetrical arguments, we obtain that we cannot move B to the right, as $B \prec B''$. We conclude that we cannot wiggle B.

Unwigglable implies having a predecessor and a successor. We prove this direction by contraposition. We first assume that B has no predecessor (however it may have successors C). We argue that B can be wiggled by moving it to the left. From the definition of a race, we obtain that:

- Since ρ is a padded timed run, the first delay d_1 of ρ is non-zero.
- For each action i' before some action $i \in B$ such that $i' \notin B$, the delay $d_{i'}$ between i' and i must be non-zero (as B has no predecessor).
- The timer fate γ_B of B is either $\perp$ or $\times$. Indeed, assume by contradiction that $\gamma_B = \bullet$. Recall that, by definition of a padded run, timers cannot have a zero value at the end of a run. Therefore, there must exist a block B'' that discards the timer of B while its valuation is zero. Thus, we have that $B'' \prec B$ which is not possible.

From these observations, we conclude that there is enough room to move B to the left. Indeed, it is possible to choose some $\epsilon < 0$ with $|\epsilon|$ small enough, such that $d_1 + \epsilon > 0$, $d_{i'} + \epsilon > 0$ for all the delays $d_{i'}$ mentioned above, and in a way that if $\gamma_B = \times$ then the new timer fate of B is still equal to $\times$. In this way we produce a timed run ρ' such that $\mathit{untime}(\rho) = \mathit{untime}(\rho')$.
It remains to explain that the blocks C participating in a race with B in ρ no longer participate in such a race in ρ'. Let C be one of these blocks, hence $B \prec C$. We have again two cases:

- There exist $i \in B$ and $i' \in C$ such that i occurs before i' in ρ and the total delay between them is zero. In the timed run ρ', this delay becomes equal to $-\epsilon > 0$ and B, C no longer participate in a race.
- The timer fate γ_C of C is equal to $\bullet$ and the timer of C is discarded by B along ρ. In ρ', we get that $\gamma_C = \times$ and B, C no longer participate in a race.

It follows that if B has no predecessor, we can wiggle it. With symmetrical arguments, if B has no successor, we can also wiggle it. Hence, the lemma holds. □

Now, we proceed to proving Proposition 9.6.9, which we restate one more time.

Proposition 9.6.9. *Let $\mathcal{M}$ be an MMT and $\rho \in \mathit{ptruns}(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.*

Proof. We prove the equivalence by showing both directions.

Wigglable implies acyclic. We prove this direction by contraposition. Suppose that G_ρ has a cycle that we can assume to be simple, *i. e.*, there are $k > 1$ distinct blocks B_ℓ, $0 \leq \ell \leq k-1$ such that $B_\ell \prec B_{\ell+1 \bmod k}$. As every block B_ℓ has a predecessor and a successor in this cycle, we cannot wiggle B_ℓ by Lemma C.4.1. Thus, ρ cannot be wiggled as it is impossible to resolve the races in which the blocks B_ℓ participate.

Acyclic implies wigglable. Assume G_ρ is acyclic. Hence, we can compute a topological sort of G_ρ restricted to the blocks participating in the races of ρ. Let B be the greatest block with respect to this sort, *i. e.*, B has no successor and it has predecessors. By Lemma C.4.1, we can wiggle B by moving it slightly to the right, thus eliminating the races between B and all the other blocks. We thus obtain a new timed run ρ' such that $\mathit{untime}(\rho) = \mathit{untime}(\rho')$ and $G_{\rho'}$ has the same vertices as G_ρ and strictly less edges (B becomes an isolated vertex). We repeat this process until the blocks of the graph are all isolated, meaning that the resulting timed run has no races and the same untimed trace as ρ. □

C.5. Proof of Theorem 9.6.11

Theorem 9.6.11. *An MMT $\mathcal{M}$ is race-avoiding*

- *if and only if any padded timed run $\rho \in \mathit{ptruns}(\mathcal{M})$ with races can be wiggled,*
- *if and only if for any padded timed run $\rho \in \mathit{ptruns}(\mathcal{M})$, its block graph G_ρ is acyclic.*

To get more intuition about the proof of Theorem 9.6.11, we recommend reading Section C.6 first.

For the first equivalence, we only need to prove one implication of this result, as by definition of wiggling, an MMT $\mathcal{M}$ is race-avoiding if all its padded timed runs with races are wigglable. The second equivalence is then a consequence of Proposition 9.6.9.

Proposition 9.6.9. Let $\mathcal{M}$ be an MMT and $\rho \in \mathit{ptruns}(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.

For this purpose, we need to introduce some new notions. Given a padded timed run

$$\begin{aligned}\rho = (q_0, \kappa_0) \xrightarrow{d_1} (q_0, \kappa_0 - d_1) \xrightarrow{i_1} (q_1, \kappa_1) \xrightarrow{d_2} \cdots \\ \xrightarrow{i_n} (q_n, \kappa_n) \xrightarrow{d_{n+1}} (q_n, \kappa_n - d_{n+1}) \in \mathit{ptruns}(\mathcal{M}),\end{aligned}$$

we extend it with additional transitions indicating when a timer has been discarded in the following way.[5] Let

$$(q_{\ell-1}, \kappa_{\ell-1}) \xrightarrow{d_\ell} (q_{\ell-1}, \kappa_{\ell-1} - d_\ell) \xrightarrow{i_\ell} (q_\ell, \kappa_\ell)$$

be a sub-run of ρ such that the set $D = \{y_1, \dots, y_m\}$ of timers discarded by i_ℓ is not empty. Then, we insert the following transitions before the next delay

5: In Section 9.6.3, this was done in the modified region automaton $\mathcal{R}(\mathcal{M})$ of $\mathcal{M}$ with the new symbols $\mathit{di}[x]$ indicating that the timer x was discarded when its value was zero. We here also consider the case when x is discarded with a non-zero value.

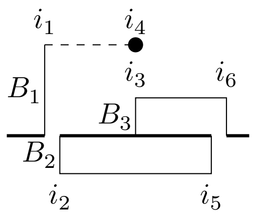


Figure C.4: The extended run of π and its block decomposition.

$d_{\ell+1}$:

$$(q_\ell,\kappa_\ell) \xrightarrow{0} (q_\ell,\kappa_\ell) \xrightarrow{j_1} (q_\ell,\kappa_\ell) \xrightarrow{0} (q_\ell,\kappa_\ell) \xrightarrow{j_2} \cdots \xrightarrow{j_m} (q_\ell,\kappa_\ell)$$

such that

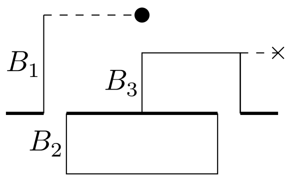
- for all k, $1 \leq k \leq m$, if y_k was discarded by i_ℓ when its value was zero, then $j_k = \bullet$, otherwise $j_k = \times$,
- each delay is zero, and
- each update is $\perp$.

We denote by $\mathrm{ext}(\rho)$ the resulting extended run, such that symbols $\bullet$ and $\times$ are also called actions.

Example C.5.1. Let us come back to the timed run π of Example 9.2.7:

$$\begin{aligned}\pi = (q_0,\emptyset) &\xrightarrow{0.5} (q_0,\emptyset) \xrightarrow[(x_1,1)]{i/o} (q_1,x_1=1) \xrightarrow{0} (q_1,x_1=1)\\ &\xrightarrow[(x_2,2)]{i/o'} (q_2,x_1=1,x_2=2) \xrightarrow{1} (q_2,x_1=0,x_2=1)\\ &\xrightarrow[(x_1,1)]{i/o} (q_2,x_1=1,x_2=1) \xrightarrow{1} (q_2,x_1=0,x_2=0)\\ &\xrightarrow[\perp]{to[x_2]/o} (q_1,x_1=0) \xrightarrow{0} (q_1,x_1=0)\\ &\xrightarrow[(x_1,1)]{to[x_1]/o} (q_1,x_1=1) \xrightarrow{0.5} (q_1,x_1=0.5).\end{aligned}$$

Its block decomposition is given in Figure 9.2c and is repeated in the margin. Its transition



$$\begin{aligned}(q_2,x_1=1,x_2=2) &\xrightarrow{1} (q_2,x_1=0,x_2=1)\\ &\xrightarrow[(x_1,1)]{i} (q_2,x_1=1,x_2=1)\end{aligned}$$

discards timer x_1 when its value is zero. Therefore, in $\mathrm{ext}(\pi)$, we insert

$$\begin{aligned}(q_2,x_1=1,x_2=1) &\xrightarrow{0} (q_2,x_1=1,x_2=1)\\ &\xrightarrow{\bullet} (q_2,x_1=1,x_2=1)\end{aligned}$$

This extended run and its block decomposition are depicted in Figure C.4, such that $i_1, i_2, \dots, i_7$ is the sequence of actions along $\mathrm{ext}(\pi)$ with $i_4 = \bullet$.

Let ρ be a padded timed run and $\operatorname{ext}(\rho)$ be its extended run. Given two actions i and i' of $\operatorname{ext}(\rho)$, the *relative elapsed time* between i and i', denoted by $\operatorname{reltime}_\rho(i, i')$, is defined as follows from the sum d of all delays between i and i' in $\operatorname{ext}(\rho)$:

- if i occurs before i', then $\operatorname{reltime}_\rho(i, i') = d$,
- otherwise $\operatorname{reltime}_\rho(i, i') = -d$.

Notice that the relative elapsed time is sensitive to the order of the actions along $\operatorname{ext}(\rho)$, and if i and i' participate in a race, then $\operatorname{reltime}_\rho(i, i') = \operatorname{reltime}_\rho(i', i) = 0$. We naturally lift this definition to a sequence of actions $i_1, i_2, \dots, i_k$ as

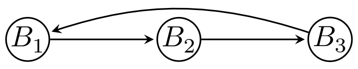
$$\operatorname{reltime}_\rho(i_1, i_2, \dots, i_k) = \sum_{\ell=1}^{k-1} \operatorname{reltime}_\rho(i_\ell, i_{\ell+1}).$$

The following lemma is trivial, as the relative elapsed time between two actions has a sign that depends on the relative position of the actions. It is illustrated by Example C.5.3 below.

Lemma C.5.2. *If the sequence $i_1, i_2, \dots, i_k$ is a cycle (i.e., $k \geq 3$ and $i_k = i_1$), then* $\operatorname{reltime}_\rho(i_1, i_2, \dots, i_k) = 0$.

Example C.5.3. We consider again the timed run π and its extended run $\operatorname{ext}(\pi)$. Recall that π cannot be wiggled as G_π is cyclic (see Figure 9.8b, which is repeated in the margin). From this cycle and the block decomposition of $\operatorname{ext}(\pi)$ (see Figure C.4), we extract the following sequence of actions: $i_1, i_2, i_5, i_6, i_3, i_4, i_1$. Notice that it is a cycle such that any two consecutive actions are either in the same block, or participate in a race, and are enumerated in a way to "follow" the cycle $B_1 \prec B_2 \prec B_3 \prec B_1$ of G_π. For instance, the first two actions i_1, i_2 describes the race $B_1 \prec B_2$, then i_2, i_5 both belong to B_2, then i_5, i_6 describes the race $B_2 \prec B_3$, etc. We have

$$\operatorname{reltime}_\pi(i_1, i_2, i_5, i_6, i_3, i_4, i_1) = 0 + 2 + 0 - 1 + 0 - 1 = 0.$$



We now proceed to the proof of Theorem 9.6.11, which we repeat again.

Theorem 9.6.11. *An MMT $\mathcal{M}$ is race-avoiding*

- *if and only if any padded timed run $\rho \in ptruns(\mathcal{M})$ with races can be wiggled,*
- *if and only if for any padded timed run $\rho \in ptruns(\mathcal{M})$, its block graph G_ρ is acyclic.*

Proof. The second equivalence holds by Proposition 9.6.9. Let us focus on proving that $\mathcal{M}$ is race-avoiding if and only if any padded timed run $\rho \in ptruns(\mathcal{M})$ is wigglable. As, by definition, it is obvious that $\mathcal{M}$ is race-avoiding if any padded timed run is wigglable, we show the other direction. Towards a contradiction, assume $\mathcal{M}$ is race-avoiding and there exists $\rho_1 \in ptruns(\mathcal{M})$ with races that is not wigglable. Since $\mathcal{M}$ is race-avoiding, there exists another padded timed run ρ_2 without races and such that $untime(\rho_1) = untime(\rho_2)$. We consider the two extended runs $\operatorname{ext}(\rho_1)$ and $\operatorname{ext}(\rho_2)$.

Proposition 9.6.9. Let $\mathcal{M}$ be an MMT and $\rho \in ptruns(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.

By Proposition 9.6.9, there must exist a cycle $\mathcal{C}$ in the block graph of ρ_1. We assume that $\mathcal{C}$ is as described in Corollary 9.6.10 and we study it on $\mathrm{ext}(\rho_1)$ (instead of ρ_1). That is, $\mathcal{C}$ is composed of $k > 1$ distinct blocks B_ℓ, $0 \leq \ell \leq k-1$, such that $B_\ell \prec B_{\ell+1 \bmod k}$, and

- any block B_ℓ participates in exactly two races described by $\mathcal{C}$,
- for any race described by $\mathcal{C}$, exactly two blocks participate in the race,
- the blocks of the cycle have at least two actions (of which one can be ●).

Corollary 9.6.10. Let $\mathcal{M}$ be an MMT and $\rho \in \mathit{ptruns}(\mathcal{M})$ be a padded timed run with races. Suppose that G_ρ is cyclic. Then there exists a cycle $\mathcal{C}$ in G_ρ such that

- any block of $\mathcal{C}$ participates in exactly two races described by this cycle,
- for any race described by $\mathcal{C}$, exactly two blocks of $\mathcal{C}$ participate in the race,
- the blocks $B = (k_1 \ldots k_m, \gamma)$ of $\mathcal{C}$ satisfy either $m \geq 2$, or $m = 1$ and $\gamma = $ ●.

We thus have the following sequence of actions from $\mathrm{ext}(\rho_1)$

$$\mathcal{S}_1 = i'_0, i_1, i'_1, \ldots, i_{k-1}, i'_{k-1}, i_0, i'_0$$

that is a cycle and such that for all ℓ, $0 \leq \ell < k$ (see also Example C.5.3):

- i_ℓ and i'_ℓ are the two symbols of B_ℓ that participate in (different) races of $\mathcal{C}$,
- $i'_\ell \in B_\ell$ and $i_{\ell+1 \bmod k} \in B_{\ell+1 \bmod k}$ participate in a race of $\mathcal{C}$, *i.e.*, $\mathrm{reltime}_{\rho_1}(i'_\ell, i_{\ell+1 \bmod k}) = 0$,
- i'_ℓ occurs before $i_{\ell+1 \bmod k}$ in $\mathrm{ext}(\rho_1)$ (since $B_\ell \prec B_{\ell+1 \bmod k}$).

By Lemma C.5.2, we have $\mathrm{reltime}_{\rho_1}(\mathcal{S}_1) = 0$. Therefore,

$$\mathrm{reltime}_{\rho_1}(\mathcal{S}_1) = \sum_{\ell=0}^{k-1} \mathrm{reltime}_{\rho_1}(i_\ell, i'_\ell) = 0. \tag{C.5.i}$$

Let us now study $\mathrm{ext}(\rho_2)$ knowing that $\mathit{untime}(\rho_1) = \mathit{untime}(\rho_2)$. Both padded timed runs ρ_1 and ρ_2 (and thus $\mathrm{ext}(\rho_1)$ and $\mathrm{ext}(\rho_2)$) must have the same block decomposition. Indeed recall that $\mathcal{M}$ is deterministic and we see the same actions. Hence, it must be that ρ_1 and ρ_2 follow the same transitions, with the same updates alongside both runs. We then have the same sequences of triggered actions, and therefore the same block decomposition in both runs.
We can thus consider the blocks of $\mathcal{C}$ seen as blocks in $\mathrm{ext}(\rho_2)$, and the sequence

$$\mathcal{S}_2 = j'_0, j_1, j'_1 \ldots, j_{k-1}, j'_{k-1}, j_0, j'_0$$

from $\mathrm{ext}(\rho_2)$ that corresponds to the sequence $\mathcal{S}_1$ from $\mathrm{ext}(\rho_1)$. We have the following properties for all $\ell, 0 \leq \ell < k$:

- if $i_\ell \in \hat{I}$ (resp. $i'_\ell \in \hat{I}$), then $j_\ell = i_\ell$ ($j'_\ell = i'_\ell$),
- if $i_\ell = $ ● (resp. $i'_\ell = $ ●), then $j_\ell = \times$ ($j'_\ell = \times$), as ρ_2 has no races and $\mathit{untime}(\rho_1) = \mathit{untime}(\rho_2)$,
- if $i_\ell, i'_\ell \in \hat{I}$, then $\mathrm{reltime}_{\rho_1}(i_\ell, i'_\ell) = \mathrm{reltime}_{\rho_2}(j_\ell, j'_\ell)$, as $i_\ell = j_\ell, i'_\ell = j'_\ell$ are in the same block in both $\mathrm{ext}(\rho_1)$ and $\mathrm{ext}(\rho_2)$,
- if one among i_ℓ, i'_ℓ is equal to ●, then

  $$|\mathrm{reltime}_{\rho_1}(i_\ell, i'_\ell)| > |\mathrm{reltime}_{\rho_2}(j_\ell, j'_\ell)|$$

  as the corresponding action in $\mathrm{ext}(\rho_2)$ is equal to $\times$ (the timer has been discarded earlier in ρ_2 than in ρ_1),
- we have $\mathrm{reltime}_{\rho_2}(j'_\ell, j_{\ell+1 \bmod k}) \neq 0$, as ρ_2 has no races,

- moreover, $\mathrm{reltime}_{\rho_2}(j'_\ell, j_{\ell+1 \bmod k}) > 0$ because i'_ℓ occurs before $i_{\ell+1 \bmod k}$ in $\mathrm{ext}(\rho_1)$ and $\mathit{untime}(\rho_1) = \mathit{untime}(\rho_2)$.

Let us first assume that no action ● ever appears in $\mathcal{S}_1$. Hence, there is no action $\times$ in $\mathcal{S}_2$. As $\mathcal{S}_2$ is a cycle, by Lemma C.5.2, we have that $\mathrm{reltime}_{\rho_2}(\mathcal{S}_2) = 0$. It follows by (C.5.i) that

$$\begin{aligned}
0 &= \mathrm{reltime}_{\rho_2}(\mathcal{S}_2) && \text{(C.5.ii)}\\
&= \sum_{\ell=0}^{k-1} \mathrm{reltime}_{\rho_2}(j'_\ell, j_{\ell+1 \bmod k}) + \sum_{\ell=0}^{k-1} \mathrm{reltime}_{\rho_2}(j_\ell, j'_\ell)\\
&> 0 + \sum_{\ell=0}^{k-1} \mathrm{reltime}_{\rho_1}(i_\ell, i'_\ell) && \text{(C.5.iii)}\\
&= \mathrm{reltime}_{\rho_1}(\mathcal{S}_1) = 0.
\end{aligned}$$

This leads to a contradiction.
Let us now assume that there exists at least one action ● in $\mathcal{S}_1$. Consider any ℓ, $0 \le \ell < k$, such that one action among i_ℓ, i'_ℓ is equal to ●. Necessarily, $i_\ell =$ ● and i'_ℓ occurs before i_ℓ in ρ_1, that is as $\mathrm{reltime}_{\rho_1}(i_\ell, i'_\ell) < 0$. Indeed, given the two races $B_{\ell-1 \bmod k} \prec B_\ell \prec B_{\ell+1 \bmod k}$, ● participates in the first race and appears at the end of B_ℓ. It follows that

$$\mathrm{reltime}_{\rho_1}(i_\ell, i'_\ell) < \mathrm{reltime}_{\rho_2}(j_\ell, j'_\ell) < 0.$$

Therefore, we get the same inequalities as in (C.5.ii), leading again to a contradiction. This completes the proof. □

C.6. Proof of Proposition 9.6.13

Proposition 9.6.13. *Let $\mathcal{M}$ be an MMT and $\mathcal{R}(\mathcal{M})$ be its modified region automaton. We can construct an MSO formula Φ of size linear in Σ and X such that a word labeling a run ρ of $\mathcal{R}(\mathcal{M})$ satisfies Φ if and only if ρ is a padded run that cannot be wiggled. Moreover, the formula Φ, in prenex normal form, has three quantifier alternations.*

Before giving the proof, some definitions are in order. Sets of finite words[6] over an alphabet Σ can be defined by sentences in MSO with the signature $(<, \{Q_a\}_{a\in\Sigma})$. Intuitively, we interpret the formula over the word $w \in \Sigma^*$ with variables being positions that take values in $\mathbb{N}$, that can be ordered with $<$, and the predicates $Q_a(p)$ indicating whether the p-th symbol of the word (structure) is a. The formulas also use variables P being sets of positions, and $P(p)$ meaning that p is a position belonging to P. We recall that a formula is in *prenex normal form* if it can be written as $Q_1 v_1 Q_2 v_2 \dots Q_n v_n F$ with F a formula without quantifiers, Q_i a quantifier and v_i a variable for all $1 \le i \le n$. We suppose the reader is familiar with the rules to put a formula into a prenex normal form. By *quantifier alternations*, we mean alternating blocks of existential or universal quantifiers, respectively denoted by $\exists^*$ and $\forall^*$.

6: Word structures, to be precise

Moreover, recall that the modifications applied on the region automaton imply the following property (see Section 9.6.3). Given a timed run ρ of an MMT, by Lemma 9.5.4, there exists an equivalent run ρ' in $\mathcal{R}(\mathcal{M})$ such that any x-block $(i_{k_1} \dots i_{k_m}, \gamma)$ of ρ is translated into the sequence of symbols $(i'_{k_1}, \dots, i'_{k_m}, \gamma')$ in ρ' with an optional symbol γ' such that:

- $i'_{k_\ell} = (i_{k_\ell}, x)$, for $1 \leq \ell < m$,
- $i'_{k_m} = (i_{k_m}, \bot)$ if $\gamma = \bot$, and (i_{k_m}, x) otherwise,
- $\gamma' = di[x]$ if $\gamma = \bullet$, and γ' does not exist otherwise.

In the sequel, we again call $\mathcal{R}(\mathcal{M})$ the modified region automaton.

We are going to describe a formula Φ such that a word labeling a run ρ of $\mathcal{R}(\mathcal{M})$ satisfies Φ if and only if ρ is a padded run that cannot be wiggled. To define Φ, we use Proposition 9.6.9. Recall that it characterizes an unwigglable run ρ by a cyclic block graph G_ρ. We also focus on the particular cycle of G_ρ as described in Corollary 9.6.10. Step by step, we create MSO formulas expressing the following statements about a run ρ of $\mathcal{R}(\mathcal{M})$:

1. Two symbols belong to the same block (see the above property about the translation of x-blocks in $\mathcal{R}(\mathcal{M})$ and the particular case of zero-valued timers that are discarded).
2. Two blocks participate in a race. Rather, we express that two symbols, one in each block, participate in a race.
3. The run is a padded run that cannot be wiggled. Rather, we express that there exists a cycle in G_ρ whose form is as in Corollary 9.6.10.

Lemma 9.5.4. Let $\mathcal{M}$ be an MMT and $\mathcal{R}(\mathcal{M})$ be its region automaton. For a timer $x \in X$, c_x denotes the largest constant to which x is updated in $\mathcal{M}$. Let $C = \max_{x \in X} c_x$. Then, the number of states of $\mathcal{R}(\mathcal{M})$ is bounded by

$$|Q^{\mathcal{M}}| \cdot |X|! \cdot 2^{|X|} \cdot (C+1)^{|X|}.$$

Moreover, for all $q, q' \in Q^{\mathcal{M}}, \kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q)), \kappa' \in \mathsf{Val}(\chi^{\mathcal{M}}(q'))$, there exists a timed run from (q, κ) to (q', κ') in $\mathcal{M}$ if and only if there exists a run from $[\![(q,\kappa)]\!]_{\cong}$ to $[\![(q',\kappa')]\!]_{\cong}$ in $\mathcal{R}(\mathcal{M})$.

Proposition 9.6.9. Let $\mathcal{M}$ be an MMT and $\rho \in \mathit{ptruns}(\mathcal{M})$ be a padded timed run with races. Then, ρ can be wiggled if and only if G_ρ is acyclic.

Corollary 9.6.10. Let $\mathcal{M}$ be an MMT and $\rho \in \mathit{ptruns}(\mathcal{M})$ be a padded timed run with races. Suppose that G_ρ is cyclic. Then there exists a cycle $\mathcal{C}$ in G_ρ such that

- any block of $\mathcal{C}$ participates in exactly two races described by this cycle,
- for any race described by $\mathcal{C}$, exactly two blocks of $\mathcal{C}$ participate in the race,
- the blocks $B = (k_1 \dots k_m, \gamma)$ of $\mathcal{C}$ satisfy either $m \geq 2$, or $m = 1$ and $\gamma = \bullet$.

C.6.1. Some useful predicates

We define four predicates to help us write the MSO formulas. The formula $\mathrm{First}(p, P)$ expresses that a position p is the first element of a set P, while $\mathrm{Last}(p, P)$ states that p is the last element of P. Finally, $\mathrm{Next}(p, P, q)$ expresses that q is the successor of p in P with regards to $<$. More formally,

$$\mathrm{First}(p, P) := P(p) \land \forall q \colon q < p \to \neg P(q) \tag{C.6.i}$$

$$\mathrm{Last}(p, P) := P(p) \land \forall q \colon q > p \to \neg P(q) \tag{C.6.ii}$$

$$\mathrm{Next}(p, P, q) := p < q \land P(p) \land P(q) \land \forall r \colon p < r < q \to \neg P(r). \tag{C.6.iii}$$

The last useful predicate, $\mathrm{Partition}(P, P_1, P_2)$, states that there exist sets of positions P, P_1, P_2 such that $P = P_1 \uplus P_2$, the first position of P is in P_1, the last one is in P_2, and the positions of P alternate between P_1 and P_2:

$$\begin{aligned} \mathrm{Partition}(P, P_1, P_2) := {} & \forall r : P(r) \leftrightarrow (P_1(r) \lor P_2(r)) \\ & \land \exists p, q : \mathrm{First}(p, P) \land \mathrm{Last}(q, P) \land P_1(p) \land P_2(q) \\ & \land \forall r : P_1(r) \leftrightarrow \neg P_2(r) \\ & \land \forall r, s : \mathrm{Next}(r, P, s) \to (P_1(r) \leftrightarrow P_2(s)). \end{aligned} \tag{C.6.iv}$$

C.6.2. Two symbols belong to the same block

We give here an MSO formula expressing that two positions $p < q$ are labeled by symbols belonging to the same x-block. Thus, the formula $\mathrm{Block}_x(p, q, P)$, with P a set of positions labeled by consecutive symbols of an x-block, states that P must respect the following constraints:

- We have $p < q$, with $p, q \in P$.
- The position p is labeled by either $(i, x) \in \Sigma$ (meaning we start an x-block B), or $(\mathit{to}[x], x) \in \Sigma$ (meaning we are in the block B),
- The input at position q is either $(\mathit{to}[x], x)$ (we are in the block B), or $(\mathit{to}[x], \bot)$ (we are at the end of B and we do not restart x), or $\mathit{di}[x]$ (we finish B by discarding its timer while its value is zero),
- Every other position $r \in P$ such that $p < r < q$ is labeled by $(\mathit{to}[x], x)$ (we restart x to keep the block active),
- There is no position $r \notin P$ between p and q such that r is labeled by some symbol of Σ among $(\mathit{to}[x], \cdot)$, $(\cdot, x)$, or $\mathit{di}[x]$ (the $\cdot$ indicates "any value"). That is, any intermediate position cannot affect x.

Formally, we have:

$$\begin{aligned}\mathsf{Affects}_x(r) := {} & Q_{(\mathit{to}[x],x)}(r) \lor Q_{(\mathit{to}[x],\bot)}(r) \\ & \lor \bigvee_{i \in I} Q_{(i,x)}(r) \lor Q_{\mathit{di}[x]}(r)\end{aligned} \tag{C.6.v}$$

$$\begin{aligned}\mathsf{Block}_x(p,q,P) := {} & p < q \land P(p) \land P(q) \\ & \land \Big(\bigvee_{i \in I} Q_{(i,x)}(p) \lor Q_{(\mathit{to}[x],x)}(p)\Big) \\ & \land \big(Q_{(\mathit{to}[x],x)}(q) \lor Q_{(\mathit{to}[x],\bot)}(q) \lor Q_{\mathit{di}[x]}(q)\big) \\ & \land \forall r\colon (p < r < q \land P(r)) \to Q_{(\mathit{to}[x],x)}(r) \\ & \land \forall r\colon (p < r < q \land \neg P(r)) \to \neg \mathsf{Affects}_x(r).\end{aligned} \tag{C.6.vi}$$

C.6.3. Two symbols participate in a race

The formula $\mathsf{Race}(p, q)$ states that two positions $p < q$ are labeled by symbols that participate in a race, *i.e.*, there is no position labeled by τ between p and q.

$$\mathsf{Race}(p,q) := p < q \land \neg(\exists r\colon p < r < q \land Q_\tau(r)). \tag{C.6.vii}$$

C.6.4. The run is unwigglable

Finally, we give a formula Φ that expresses that a word is the label of a padded run ρ that cannot be wiggled, *i.e.*, that highlights a cycle (as in Corollary 9.6.10) in the block graph of ρ. The idea of that formula is as follows. There are positions $p_1 < q_1 < p_2 < q_2 < \cdots < p_m < q_m$ such that each pair p_k, q_k participate in a race. Moreover, for any q_k belonging to a block B_k, there must exist a p_ℓ that also belongs to B_k. Notice that p_ℓ is not necessarily after q_k. See Figure 9.9 for an illustration of that scenario with $m = 5$.

The formula Φ states that there exist sets of positions P, P_1, P_2 such that:

- We have $P_1 \uplus P_2$ forming a partition of P as described previously,
- For any $p \in P_1$ and $q \in P_2$ such that q is the next element after p in P, we have $\mathsf{Race}(p, q)$.

- For any $q \in P_2$, there must exist a p in P_1 such that p and q belong to the same block. That is, there must exist a timer x and a set P' such that $\mathrm{Block}_x(p, q, P')$ if $p < q$ or $\mathrm{Block}_x(q, p, P')$ if $p > q$.

Finally, in order to describe a padded run ρ, the first and last positions of the word must be labeled with τ (*i.e.*, a non-zero delay). These positions do not belong to P.

$$\begin{aligned}\mathrm{InBlock}(p, q, P') := {} & (p < q \wedge \bigvee_x \mathrm{Block}_x(p, q, P')) \\ & \vee (p > q \wedge \bigvee_x \mathrm{Block}_x(q, p, P'))\end{aligned} \tag{C.6.viii}$$

We now have all the pieces needed to give Φ.

$$\begin{aligned}\Phi := {} & \exists P, P_1, P_2 \colon \Big(\mathrm{Partition}(P, P_1, P_2) \\ & \wedge \forall p, q \colon (P_1(p) \wedge \mathrm{Next}(p, P, q)) \to \mathrm{Race}(p, q) \\ & \wedge \forall q \colon P_2(q) \to (\exists p, P' \colon P_1(p) \wedge \mathrm{InBlock}(p, q, P'))\Big) \\ & \wedge Q_\tau(1) \wedge (\exists r \colon Q_\tau(r) \wedge (\forall r' \colon r' < r)).\end{aligned} \tag{C.6.ix}$$

C.6.5. Correctness

Now that we have constructed the formula Φ, let us show that it correctly encodes that the word labeling a run ρ in $\mathcal{R}(\mathcal{M})$ satisfies Φ if and only if ρ is a padded run that is not wigglable.

Run satisfies Φ implies unwigglable/ Assume ρ in $\mathcal{R}(\mathcal{M})$ is a padded run that is not wigglable. Let $w \in \Sigma^*$ be its labeling. By Proposition 9.6.9, we know that the block graph G_ρ is cyclic. Moreover, by Corollary 9.6.10, there exists a cycle whose blocks satisfy the following properties: exactly two blocks participate in any race of the cycle, any block participates in exactly two races, and any block has a size at least equal to two.[7] We consider this particular cycle $(B_0, \dots, B_{m-1}, B_0)$.

7: Recall the way blocks in $\mathcal{A}$ are translated into blocks in $\mathcal{R}(\mathcal{M})$.

From the races in which the blocks B_k participate, we define the sets P_1, P_2 of positions, and, thus, $P = P_1 \uplus P_2$ as follows. For every $B_k \prec B_{k+1 \bmod m}$ in the cycle, consider $a \in B_k$ and $b \in B_{k+1 \bmod m}$ that are the two symbols of w participating in a race. We add the position of a in P_1 and the position of b in P_2 (see Figure 9.9 to get intuition). Thus, the positions in P alternate between P_1 and P_2, the first element of P is in P_1, and the last is in P_2. Moreover, for any $p \in P_1$ and $q \in P_2$ such that q is the successor of p in P, it holds that $\mathrm{Race}(p, q)$. That is, the second line of formula (C.6.ix) about races is satisfied by w.

We have to show that the third line of (C.6.ix) is also satisfied by w, *i.e.*, for any position q in P_2, there exists a position p in P_1 such that p and q belong to the same block. Let $q \in P_2$. By construction, q belongs to some block B_k of the cycle. By Corollary 9.6.10, B_k participate in exactly two races of the cycle, one as described above with $\mathrm{Race}(p, q)$, and another one with $\mathrm{Race}(p', q')$ for some other positions $p' \in P_1$ and $q' \in P_2$. Necessarily, p' is a position of a symbol in B_k (and not q' by choice of the cycle), such that either $p' < q$ or $p' > q$. Thus, the third line of (C.6.ix) is satisfied by w.

Finally, since ρ is padded, it must be that its first and last delays are positive, *i. e.*, the corresponding positions are labeled by τ. Hence, the last line of (C.6.ix) is satisfied by w. We conclude that w satisfies all conjuncts of (C.6.ix) and then also the formula Φ.

Unwigglable implies run satisfies Φ. Assume now that the label w of a run ρ in $\mathcal{R}(\mathcal{M})$ satisfies formula Φ. Since the last line of (C.6.ix) forces the first and last symbols of w to be τ, the formula describes a padded run of $\mathcal{R}(\mathcal{M})$.

Let $P, P_1 = \{p_1, \dots, p_m\}$, and $P_2 = \{q_1, \dots, q_m\}$ be the sets satisfying the formula Φ such that $\mathrm{Next}(p_k, P, q_k)$ is satisfied for all k. Then, by (C.6.vii), it holds that $\mathrm{Race}(p_k, q_k)$ is also satisfied, *i. e.*, the symbols of w labeling the positions p_k, q_k participate in a race. The third line of (C.6.ix) implies that there are at most m blocks involved in these races. Notice that there can be less than m blocks, as for q, q' in P_2 with $q \neq q'$, we could have the same $p \in P_1$ that makes sub-formula InBlock satisfied in (C.6.ix).

From formula Φ, we are going to construct a part of the block graph G_ρ of ρ that is cyclic. We proceed inductively as follows:

- Take an arbitrary position $q_{k_0} \in P_2$. There exists $p_{k_1} \in P_1$ such that $\mathrm{Block}_{x_1}(q_{k_0}, p_{k_1}, P'_1)$ or $\mathrm{Block}_{x_1}(p_{k_1}, q_{k_0}, P'_1)$ is satisfied. Call B_1 the related x_1-block.
- Let $q_{k_1} \in P_2$ (observe that we use the index obtained at the previous step). Then, there exists a $p_{k_2} \in P_1$ such that $\mathrm{Block}_{x_2}(q_{k_1}, p_{k_2}, P'_2)$ or $\mathrm{Block}_{x_2}(p_{k_2}, q_{k_1}, P'_2)$ is satisfied. For the related x_2-block B_2, we have $B_1 \prec B_2$ as p_{k_1}, q_{k_1} participate in a race, and $p_{k_1} \in B_1, q_{k_1} \in B_2$.
- Let $q_{k_2} \in P_2$ (again, we use the index of the previous step). There exists a $p_{k_3} \in P_1$ such that $\mathrm{Block}_{x_3}(q_{k_2}, p_{k_3}, P'_3)$ or $\mathrm{Block}_{x_3}(p_{k_3}, q_{k_2}, P'_3)$ is satisfied. For the related x_3-block B_3, we have $B_2 \prec B_3$.
- We repeat this process until we obtain a cycle. This situation necessarily arises as the number of blocks is bounded by m.

This shows that G_ρ is cyclic.

C.6.6. Prenex form of the formula

Finally, let us write the formula under prenex normal form.

Formula $\mathrm{Block}_x(p, q, B)$, see (C.6.vi), can be easily rewritten in prenex normal form that starts with a block $\forall^*$ of universal quantifiers. Similarly $\mathrm{Race}(p, q)$, see (C.6.vii), requires a single universal quantifier. Let us consider the formula Φ putting aside the quantifiers $\exists P, P_1, P_2$, see (C.6.ix). Notice that formulas (C.6.i) to (C.6.iii) all use a single universal quantifier. The first conjunct of (C.6.ix) uses $\mathrm{Partition}(P, P_1, P_2)$, see (C.6.iv), that can be rewritten with three blocks $\forall^*\exists^*\forall^*$. The second (resp. last) conjunct can be rewritten with two blocks $\forall^*\exists^*$ (resp. $\exists^*\forall^*$), and the third one with three blocks $\forall^*\exists^*\forall^*$. Hence, we obtain that the quantifiers of the prenex normal form of Φ are $\exists^*\forall^*\exists^*\forall^*$, *i. e.*, with three quantifier alternations, as expected.

Finally, by carefully examining the formulas, we notice that most of them have constant size, except the formulas $\mathrm{Affects}_x(r)$ and $\mathrm{Block}_x(p, q, P)$ whose sizes are linear in $|I|$, and the formulas $\mathrm{InBlock}(p, q, P')$ and Φ whose sizes are linear in $|I|$ and $|X|$.

C.7. Proof of Theorem 9.6.4

Theorem 9.6.4. *Deciding whether an MMT is race-avoiding is* PSPACE*-hard and in* 3EXP*. It is in* PSPACE *if the sets of actions I and of timers X are fixed.*

We begin by proving the upper bound.

C.7.1. Upper bound

We make use of the Büchi-Elgot-Trakhtenbrot theorem: A language is regular if and only if it can be defined as the set of all words satisfied by an MSO formula (with effective translations, see [GTW03; Tho97]). First, from the formula Φ of Proposition 9.6.13, we can construct a finite-state automaton $\mathcal{N}$ whose language is the set of all words satisfying Φ. Due to the reduction from non-deterministic to deterministic automaton, each quantifier alternation induces an exponential blowup in the automaton construction. Hence, the size of $\mathcal{N}$ is triple-exponential as Φ has three quantifier alternations. Second, we compute the intersection of $\mathcal{N}$ with $\mathcal{R}(\mathcal{M})$ — itself exponential in size. This can be done in polynomial time in the sizes of both automata, *i. e.*, in 3EXP. Finally, the language of the resulting automaton is empty if and only if there is no padded run of $\mathcal{A}$ that cannot be wiggled, and this can be checked in polynomial time with respect to the size of the (triple-exponential) automaton.

[GTW03]: Grädel et al. (2003), *Automata, logics, and infinite games: a guide to current research*
[Tho97]: Thomas (1997), "Languages, Automata, and Logic"

Proposition 9.6.13. Let $\mathcal{M}$ be an MMT and $\mathcal{R}(\mathcal{M})$ be its modified region automaton. We can construct an MSO formula Φ of size linear in Σ and X such that a word labeling a run ρ of $\mathcal{R}(\mathcal{M})$ satisfies Φ if and only if ρ is a padded run that cannot be wiggled. Moreover, the formula Φ, in prenex normal form, has three quantifier alternations.

C.7.2. Lower bound

In this section, we prove that deciding whether all padded timed runs of a given MMT are wigglable is a PSPACE-hard problem. The idea of the proof is to leverage the PSPACE-hardness proof for the reachability problem, see Theorem 9.5.1. We use the same notations as in the proof provided for the latter result in Section C.3.

Let $\mathcal{A}$ be an LBTM and w be an input word. Let $\mathcal{M}_{\mathcal{A},w}$ be the MMT constructed from $\mathcal{A}$ and w in the proof of Theorem 9.5.1, such that the state r_{done} is reachable in the MMT if and only if the LBTM accepts w. We are going to slightly modify the LBTM and the constructed MMT such that any padded timed run ρ of $\mathcal{M}_{\mathcal{A},w}$ has an acyclic block graph G_ρ. Thanks to Proposition 9.6.9, this is equivalent to stating that ρ can be wiggled. Then, we give a widget that extends any padded timed run ρ reaching r_{done} into one that is unwigglable. Hence, the given LBTM accepts w if the constructed MMT has some unwigglable padded timed run.

First, we modify the LBTM $\mathcal{A}$ and word $w = w_1 \dots w_n$ to obtain a new LBTM $\mathcal{A}'$ that accepts w if and only if it does so while maintaining the invariant that

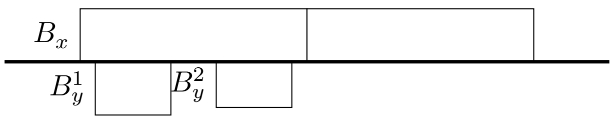


Figure C.5: Visualization of the forced buffer using timer y.

no two cells of the tape of $\mathcal{A}'$ hold the same symbol. Let Σ be the alphabet of $\mathcal{A}$. We create a new word $w' = (w_1, 1)(w_2, 2) \dots (w_n, n)$, and a new LBTM $\mathcal{A}'$ over the alphabet $\Sigma' = \Sigma \times \{1, \dots, n\}$ such that $\mathcal{A}'$ simulates $\mathcal{A}$ by discarding the second component of each symbol of Σ', except that whenever $\mathcal{A}$ writes the symbol a at the position i on the tape, $\mathcal{A}'$ writes the symbol (a, i). This requires to store the current i in the state of $\mathcal{A}'$, inducing a polynomial blowup in n for the number of states of $\mathcal{A}'$. Thanks to the second component, we indeed have that every cell contains a symbol that is different from any other cell.

Second, we modify the construction of the MMT $\mathcal{M}_{\mathcal{A}',w'}$ as follows. We add a new timer y which we use to force a block acting as a buffer before and after each action implying the timer x when the value of *clock* is equal to zero, as illustrated in Figure C.5. Therefore, any input-action must take place between these two buffers. In order to have enough room for these buffers, we multiply by three all the values at which the timers $x, x_1, \dots, x_n$ are updated (in particular x is always (re)started with value 3 instead of 1). For the initialization of the timers $x, x_1, \dots, x_n$ (at the start of phase 1), we add new states $\langle r_1, y_k \rangle$ and $\langle q_0, 1, 0, 0, y_k \rangle$, for $k = 1, 2$, and modify the transitions to force y to start and time out, then we force the initialization part and, finally, force y to start and time out again:

$$r_0 \xrightarrow[(x,3)]{go} \langle r_1, y_1 \rangle \xrightarrow[(y,1)]{go} \langle r_1, y_2 \rangle \xrightarrow[\perp]{to[y]} r_1 \xrightarrow{go} \dots \xrightarrow{go} r_n$$
$$\xrightarrow[(x_n,6j)]{go} \langle q_0, 1, 0, 0, y_1 \rangle \xrightarrow[(y,1)]{go} \langle q_0, 1, 0, 0, y_2 \rangle \xrightarrow[\perp]{to[y]} \langle q_0, 1, 0, 0 \rangle$$

with $6j$ the value at which x_n must be set (see the proof of Theorem 9.5.1). We define

$$\chi(\langle r_1, y_1 \rangle) = \chi(r_1) = \{x\}$$
$$\chi(\langle r_1, y_2 \rangle) = \{x, y\}$$
$$\chi(\langle q_0, 1, 0, 0, y_1 \rangle) = \chi(\langle q_0, 1, 0, 0 \rangle) = \{x, x_1, \dots, x_n\}$$

and

$$\chi(\langle q_0, 1, 0, 0, y_2 \rangle) = \{x, x_1, \dots, x_n, y\}.$$

In states where $\mathit{clock} = 0$ in the other phases, we do likewise in the following way. For each state $\langle q, i, \mathit{symbol}, 0 \rangle$ of the MMT with $\mathit{symbol} > 0$, we add new states $\langle q, i, \mathit{symbol}, 0, y_k \rangle$ and $\langle q', j, 0, 0, y_k \rangle$, for $k = 1, 2$, and modify the transitions to force y to start and time out, then allow a t-transition,[8] and,

8: We use the notations of the proof of Theorem 9.5.1.

finally, force y to start and time out again:

$$\begin{aligned}\langle q, i, \mathit{symbol}, 0, y_1\rangle &\xrightarrow[(y,1)]{\mathit{go}} \langle q, i, \mathit{symbol}, 0, y_2\rangle \\ &\xrightarrow[\bot]{\mathit{to}[y]} \langle q, i, \mathit{symbol}, 0\rangle \\ &\xrightarrow[(x_i, 6j)]{t} \langle q', j, 0, 0, y_1\rangle \xrightarrow[(y,1)]{\mathit{go}} \langle q', j, 0, 0, y_2\rangle \\ &\xrightarrow[\bot]{\mathit{to}[y]} \langle q', j, 0, 0\rangle.\end{aligned}$$

Any $\mathit{to}[x]$-transition leading to $\langle q, i, \mathit{symbol}, 0\rangle$ is instead redirected to the state $\langle q, i, \mathit{symbol}, 0, y_1\rangle$. We then define

$$\chi(\langle q, i, \mathit{symbol}, 0, y_1\rangle) = \chi(\langle q', j, 0, 0, y_1\rangle) = \{x, x_1, \dots, x_n\}$$

and

$$\chi(\langle q, i, \mathit{symbol}, 0, y_2\rangle) = \chi(\langle q', j, 0, 0, y_2\rangle) = \{x, x_1, \dots, x_n, y\}.$$

Any other transition leads to r_{sink}.

Third, let $\mathcal{M}'$ be the MMT obtained with this modified construction. We now argue that any padded timed run $\rho \in \mathit{ptruns}(\mathcal{M}')$ can be wiggled. We do so by proving that the block graph of ρ is acyclic In the sequel, we say that two block B, B' are *incomparable* if neither $B \prec B'$ nor $B' \prec B$.

1. Suppose first that ρ does not contain the state r_{sink}.
   - Recall that x is never discarded and every transition reading $\mathit{to}[x]$ restarts it. Thus, there exists a single x-block; we call it B_x. By construction, none of the $\mathit{to}[y]$-transitions update y. Therefore, we have strictly more than one y-block. Call them $B_y^1, B_y^2, \dots, B_y^{m_y}$ in the order they are seen alongside ρ. For any odd i, it may be that $B_x \prec B_y^i$ (if the input-action starting y occurs at the same time x times out), $B_y^i \prec B_y^{i+1}$ (if the sum of the delays between the timeout of y in B_y^i and the go-transition starting B_y^{i+1} is zero), and $B_y^{i+1} \prec B_x$. However, it is impossible to have $B_x \prec B_y^i \prec B_y^{i+1} \prec B_x$, as x is (re)started with value 3 and there are exactly two units of time if $B_y^i \prec B_y^{i+1}$ (since y is always started with value 1). That is, we do not have a cycle. Moreover, since the y-blocks only appear when the *clock* component of the current state is zero, B_y^j and B_y^{j+1} are incomparable for every even j.
   - Let us now focus on the timers x_i. Since a t-action can discard x_i, we may have multiple x_i-blocks, say $B_{x_i}^1, \dots, B_{x_i}^{m_{x_i}}$ (again, in the order they are seen in ρ). Since x_i is updated such that it cannot time out while the *clock* component of the current state is zero (*i.e.*, the timer fate of the corresponding block is $\times$) and only one t-transition can occur per phase of $\mathcal{A}'$, all of the x_i-blocks are pairwise incomparable. Moreover, thanks to the y-buffers, B_x and any block $B_{x_i}^j$ are incomparable. As stated before, it may happen that a t-transition of some block $B_{x_i}^j$ occurs concurrently with an action of a block B_y^ℓ (resp. $B_y^{\ell+1}$) with ℓ an odd number. By construction, $B_y^\ell \prec B_{x_i}^j$ (resp. $B_{x_i}^j \prec B_y^{\ell+1}$). Note that it is possible that $B_y^\ell \prec B_{x_i}^j \prec B_y^{\ell+1}$ if all blocks participate in the same race.

- We now consider two different timers x_i and x_k. Since the LBTM $\mathcal{A}'$ is such that no two cells contain the same value, it must be that x_i and x_k are always updated to time out in states containing different *clock* component in $\mathcal{A}'$ (as, otherwise, this would imply that the two cells of $\mathcal{A}'$ contain identical symbols). That is, it is not possible for two timers to time out concurrently. However, during the initialization, it may be that the actions starting $B^1_{x_1}, B^1_{x_2}, \dots, B^1_{x_n}$ occur at the same time. We thus have $B^1_{x_1} \prec B^1_{x_2} \prec \dots \prec B^1_{x_n}$ and the blocks $B^j_{x_i}$ and $B^\ell_{x_k}$ are incomparable with $j, \ell > 1$.

Using all these facts over the races and $\prec$, we deduce that G_ρ is acyclic, *i.e.*, ρ can be wiggled.

2. Suppose now that ρ contains the state r_{sink}. Recall that $\chi(r_{sink}) = \emptyset$, meaning that the update of every transition ending in r_{sink} is $\perp$, every timer is stopped, and every new block started after reaching r_{sink} contains exactly one action. We thus focus on the prefix of the run up to the transition leading to r_{sink}. Call this last transition t^* with action i^*. By construction, it must be this prefix is a run that satisfies the constraints explained above (*i.e.*, there is no cycle in the block graph induced by the prefix of the run). Let us show that adding t^* after the prefix does not induce a cycle in the block graph.
Suppose first that i^* is an input. As the update of t^* is $\perp$, it must be that i^* is the only action of its block, and this block cannot appear in a cycle. Suppose now that i^* is a timeout-action.

   - Let us study the initialization or a simulation step (*i.e.*, the start of a phase). Recall that no timer x_i can time out, *i.e.*, we only have to consider the timers x and y. In this case, we must have $i^* = \mathit{to}[x]$. Indeed, this happens when a block B^ℓ_y is started too late, in a way that $\mathit{to}[x]$ occurs before y times out, or when such a block is not started at all. The only hope to have a cycle is to adapt to the current situation the discussion we made above about $B_x \prec B^\ell_y \prec B^{\ell+1}_y \prec B_x$ with ℓ odd. Here, as $i^* = \mathit{to}[x]$ stops the timer y and x (resp. y) is (re)started with value 3 (resp. value 1), we get $B_x \prec B^\ell_y$ and $B_x \prec B^{\ell+1}_y$. We thus have no cycle.
   - Otherwise, we consider a state in which the *clock* value is not zero (*i.e.*, this is not the start of a phase). By construction, a $\mathit{to}[x]$-, or any $\mathit{to}[x_i]$-transition cannot lead to r_{sink}, and y is never started. Thus, i^* cannot be a timeout-action in this case.

   Hence, we covered every case and never obtained a cycle, *i.e.*, ρ is wigglable.

We have thus proved that each timed run $\rho \in \mathit{ptruns}(\mathcal{M}')$ can be wiggled.

Finally, we add a widget that forces an unwigglable run after r_{done}. To do so, we add new timers z, z' and states s_1 to s_4, and define the following transitions

$$r_{done} \xrightarrow[(z,1)]{go} s_1 \xrightarrow[(z',1)]{go} s_2 \xrightarrow[\perp]{to[z']} s_3 \xrightarrow[\perp]{to[z]} s_4.$$

We define $\chi(s_1) = \chi(s_3) = \{z\}$, $\chi(s_2) = \{z, z'\}$, and $\chi(s_4) = \emptyset$. Given a padded timed run ending in r_{done}, the only ways to extend it into a padded timed run reaching s_4 are by adding the following sequence of transitions, with any $d > 0$:

$$\begin{aligned}(r_{done}, \emptyset) &\xrightarrow[(z,1)]{go} (s_1, z = 1) \xrightarrow{0} (s_1, z = 1)\\ &\xrightarrow[(z',1)]{go} (s_2, z = 1, z' = 1) \xrightarrow{1} (s_2, z = 0, z' = 0)\\ &\xrightarrow[\bot]{to[z']} (s_3, z = 0) \xrightarrow{0} (s_3, z = 0)\\ &\xrightarrow[\bot]{to[z]} (s_4, \emptyset) \xrightarrow{d} (s_4, \emptyset).\end{aligned} \tag{C.7.i}$$

The resulting padded timed run is not wigglable, as two blocks B_z and $B_{z'}$ have been added such that $B_z \prec B_{z'} \prec B_z$.

To conclude, it remains to prove that the given $\mathcal{A}'$ accepts the given w' if and only if there exists an unwigglable padded timed run in the MMT $\mathcal{M}'$ extended with the widget. First, suppose that $\mathcal{A}'$ accepts w'. Then by the proof of Theorem 9.5.1, we know that there exists a timed run $\rho \in \mathit{truns}(\mathcal{M}')$ reaching r_{done}. As both r_0 and r_{done} do not have any active timer, the first and last delays of ρ can be made positive, thus making ρ padded. We then extend ρ with (C.7.i) and obtain an unwigglable run that is still padded. Second, suppose that there exists a padded timed run ρ that cannot be wiggled. We proved above that if ρ ends in some state of $\mathcal{M}'$, then ρ is wigglable. Therefore, by construction of the widget, ρ has to end with s_4, meaning that a prefix of ρ reaches r_{done}. It follows that w' is accepted by $\mathcal{M}'$. Thus, deciding whether all padded timed runs of an MMT can be wiggled is a PSPACE-hard problem.

C.8. Proof of Theorem 9.7.5

Theorem 9.7.5. *Let $\mathcal{M}$ be a complete and sound MMT, and $\mathit{zone}(\mathcal{M})$ be its zone MMT. Then,*

- *both MMTs $\mathcal{M}$ and $\mathit{zone}(\mathcal{M})$ have the same timed behaviors, i.e., it holds that for every timed word w, state $q \in Q^{\mathcal{M}}$, and valuation $\kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q))$,*
$$(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w} (q, \kappa) \in \mathit{truns}(\mathcal{M})$$
if and only if there exists a zone Z over $\chi^{\mathcal{M}}(q)$ such that $\kappa \in Z$ and
$$((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{w} ((q, Z), \kappa) \in \mathit{truns}(\mathit{zone}(\mathcal{M})).$$
- *$\mathcal{M}$ and $\mathit{zone}(\mathcal{M})$ have the same feasible runs,*
- *$\mathit{zone}(\mathcal{M})$ is sound and complete,*
- *$\mathcal{M}$ and $\mathit{zone}(\mathcal{M})$ are symbolically equivalent, and*
- *any run of $\mathit{zone}(\mathcal{M})$ is feasible.*

We prove each statement independently and in the order given in the theorem.

Lemma C.8.1. *Let $zone(\mathcal{M})$ be the zone MMT of some sound and complete MMT $\mathcal{M}$. Then, for every timed word w, state $q \in Q^{\mathcal{M}}$, and valuation $\kappa \in$ $\mathsf{Val}(\chi^{\mathcal{M}}(q))$,*

$$(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w} (q, \kappa) \in \mathit{truns}(\mathcal{M})$$

if and only if there exists a zone Z over $\chi^{\mathcal{M}}(q)$ such that $\kappa \in Z$ and

$$((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{w} ((q, Z), \kappa) \in \mathit{truns}(zone(\mathcal{M})).$$

Proof. We focus on the $\Rightarrow$ direction. The other direction can be obtained with similar arguments. Let $q \in Q^{\mathcal{M}}, \kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q))$, and w be a timed word such that $(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w} (q, \kappa)$. We show that there exists a zone Z over $\chi^{\mathcal{M}}(q)$ such that $\kappa \in Z$ and $((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{w} ((q, Z), \kappa)$. We proceed by induction over the length of w.
Base case: $|w| = 0$, *i.e.*, $w = d$ with $d \in \mathbb{R}^{\geq 0}$. Since no timer is active, it is clear that we have the runs

$$(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{d} (q_0^{\mathcal{M}}, \emptyset) \qquad \text{and} \qquad ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{d} ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset),$$

and $\emptyset \in \{\emptyset\}$.
Induction step: let $k \in \mathbb{N}$ and assume the implication is true for every timed word of length k. Let $w = w' \cdot i \cdot d$ of length $k + 1$, *i.e.*, $|w'| = k$, $i \in A(\mathcal{M})$, and $d \in \mathbb{R}^{\geq 0}$. Then, we have

$$(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w'} (p, \lambda) \xrightarrow[u]{i} (q, \kappa) \xrightarrow{d} (q, \kappa - d).$$

This implies that $d \leq \min_{y \in \chi^{\mathcal{M}}(q)} \kappa(y)$.[9] By induction hypothesis, we have that

$$((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{w'} ((p, Z_p), \lambda)$$

such that $\lambda \in Z_p$. It is then sufficient to show that we have

$$((p, Z_p), \lambda) \xrightarrow[u]{i} ((q, Z), \kappa) \xrightarrow{d} ((q, Z), \kappa - d)$$

with $\kappa - d \in Z$.
By construction of the zone MMT and as $p \xrightarrow{i} q$ is defined in $\mathcal{M}$, the i-transition from (p, Z_p) to (q, Z) is defined if and only if Z is not empty and

$$Z = \begin{cases} (Z_p \lceil \chi^{\mathcal{M}}(q)) \downarrow & \text{if } i \in I \text{ and } u = \bot \\ ((Z_p[x = c]) \lceil \chi^{\mathcal{M}}(q)) \downarrow & \text{if } i \in I \text{ and } u = (x, c) \\ ((\mathit{to}[Z_p, x]) \lceil \chi^{\mathcal{M}}(q)) \downarrow & \text{if } i = \mathit{to}[x] \text{ and } u = \bot \\ (((\mathit{to}[Z_p, x])[x = c]) \lceil \chi^{\mathcal{M}}(q)) \downarrow & \text{if } i = \mathit{to}[x] \text{ and } u = (x, c). \end{cases}$$

Since Z_p is not empty (as $\lambda \in Z_p$) and $(p, \lambda) \xrightarrow{i}$ can be triggered (meaning that $\lambda(x) = 0$ if $i = \mathit{to}[x]$), we have that Z is also not empty. Hence, $((p, Z_p), \lambda) \xrightarrow{i} ((q, Z), \kappa)$ is well-defined and can be triggered.
Let us show that $\kappa \in Z$. By definition of a timed run, $\kappa \in \mathsf{Val}(\chi^{\mathcal{M}}(q))$.

9: When $\chi^{\mathcal{M}}(q)$ is empty, we assume that $\min_{y \in \chi^{\mathcal{M}}(q)} \kappa(y)$ is equal to $+\infty$.

Moreover, Z is a zone over $\chi^{\mathcal{M}}(q)$. We know that $\lambda \in Z_p$ and κ is constructed from λ by discarding the values for timers that are stopped by the discrete transition and, maybe, (re)starting a timer. Since Z is constructed using the same operations, it follows that $\kappa \in Z$.
Finally, we process the delay d. We already know that $d \leq \min_{y \in \chi^{\mathcal{M}}(q)} \kappa(y)$. Hence, it is feasible to wait d units of time from $((q, Z), \kappa)$. Moreover, as Z is already its downward closure, we still have that $\kappa - d \in Z$.
We thus obtain the implication. Again, one can show the other direction using similar arguments, by definition of *zone*$(\mathcal{M})$. □

From the previous lemma, we conclude that any feasible run of $\mathcal{M}$ can be reproduced in *zone*$(\mathcal{M})$ and vice-versa. Recall that $A(\mathcal{M}) = A(\mathit{zone}(\mathcal{M}))$.

Corollary C.8.2. *Let $\mathcal{M}$ be a sound and complete MMT and zone$(\mathcal{M})$ be its zone MMT. Then, for all words $w \in A(\mathcal{M})^*$, $q_0^{\mathcal{M}} \xrightarrow{w} q$ is in runs$(\mathcal{M})$ and is feasible if and only if there exists a zone Z such that $(q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{w} (q, Z)$ is in runs$(\mathit{zone}(\mathcal{M}))$ and is feasible.*

Proof. Let $q_0^{\mathcal{M}} \xrightarrow{w} q$ be a feasible run of $\mathcal{M}$. Then, there exists a timed run $(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{v} (q, \kappa)$ of $\mathcal{M}$. By Lemma C.8.1, it follows that $((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{v} ((q, Z), \kappa)$ is a timed run of *zone*$(\mathcal{M})$ for some zone Z such that $\kappa \in Z$. As v and w use the same actions, the run $(q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{w} (q, Z)$ is a feasible run of *zone*$(\mathcal{M})$. The other direction holds with similar arguments. □

From Lemma C.8.1, we also deduce that *zone*$(\mathcal{M})$ is complete.

Lemma C.8.3. *Let $\mathcal{M}$ be a sound and complete MMT and zone$(\mathcal{M})$ be its zone MMT. Then, zone$(\mathcal{M})$ is sound and complete.*

Proof. By Lemma 9.7.4, we already know that *zone*$(\mathcal{M})$ is sound. By hypothesis, $\mathcal{M}$ is complete. This means that for every $q \in Q^{\mathcal{M}}$ and $i \in A(\mathcal{M})$, we have $q \xrightarrow{i} \in \mathit{runs}(\mathcal{M})$ if and only if $i \in I \cup \mathit{TO}[\chi_0^{\mathcal{M}}(q)]$. Let us show that *zone*$(\mathcal{M})$ is complete, *i.e.*, for every $(q, Z) \in Q^{\mathit{zone}(\mathcal{M})}$ and $i \in A(\mathit{zone}(\mathcal{M})) = A(\mathcal{M})$, we have $(q, Z) \xrightarrow{i} \in \mathit{runs}(\mathit{zone}(\mathcal{M}))$ if and only if $i \in I \cup \mathit{TO}[\chi_0^{\mathit{zone}(\mathcal{M})}((q, Z))]$.
Let (q, Z) be a (reachable) state of *zone*$(\mathcal{M})$ and $i \in A(\mathcal{M})$. We have the following cases:

Lemma 9.7.4. Let $\mathcal{M}$ be a sound MMT. Then, *zone*$(\mathcal{M})$ has finitely many states and is sound.

- $i \in I$, in which case $q \xrightarrow{i} \in \mathit{runs}(\mathcal{M})$ as $\mathcal{M}$ is complete. By construction of *zone*$(\mathcal{M})$, it follows that $(q, Z) \xrightarrow{i} \in \mathit{runs}(\mathit{zone}(\mathcal{M}))$.
- $i = \mathit{to}[x]$ for some $x \in X^{\mathcal{M}} = X^{\mathit{zone}(\mathcal{M})}$. As $\mathcal{M}$ is complete, $q \xrightarrow{i} \in \mathit{runs}(\mathcal{M})$ if and only if $i \in \mathit{TO}[\chi_0^{\mathcal{M}}(q)]$. That is, $q \xrightarrow{i} \in \mathit{runs}(\mathcal{M})$ if and only if there exist a valuation κ and a timed word w such that $\kappa(x) = 0$ and $(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{w} (q, \kappa)$. Let us consider all such pairs of valuation κ and timed word w. We thus have two cases:
  - There exist some pair w, κ such that $\kappa \in Z$. By Lemma C.8.1, we obtain that $(q_0^{\mathit{zone}(\mathcal{M})}, \emptyset) \xrightarrow{w} ((q, Z), \kappa)$. Hence, $\mathit{to}[Z, x] \neq \emptyset$, the

to$[x]$-transition is defined from (q, Z), and x is enabled in (q, Z).

- Among all pairs w, κ, no κ belongs to Z. Then, we deduce that any timed run $(q_0^{\mathit{zone}(\mathcal{M})}, \emptyset) \xrightarrow{v} ((q, Z), \lambda)$ (recall that (q, Z) is reachable) is such that $\lambda(x) \neq 0$, again by Lemma C.8.1. So, $\mathit{to}[Z, x] = \emptyset$, the *to*$[x]$-transition is not defined, and x is not enabled in (q, Z).

Hence, $(q, Z) \xrightarrow{\mathit{to}[x]} \in \mathit{runs}(\mathit{zone}(\mathcal{M}))$ if and only if $x \in \chi_0^{\mathit{zone}(\mathcal{M})}((q, Z))$.

We conclude that $(q, Z) \xrightarrow{i}$ is a run of $\mathit{zone}(\mathcal{M})$ if and only if $i \in I \cup \mathit{TO}[\chi_0^{\mathit{zone}(\mathcal{M})}((q, Z))]$, *i.e.*, $\mathit{zone}(\mathcal{M})$ is complete. □

Let us now move towards proving that $\mathcal{M} \overset{\mathrm{sym}}{\approx} \mathit{zone}(\mathcal{M})$.

Lemma C.8.4. *Let $\mathcal{M}$ be a sound and complete MMT. Then, $\mathcal{M} \overset{\mathrm{sym}}{\approx} \mathit{zone}(\mathcal{M})$.*

Proof. We have to show that for every symbolic word $\mathtt{i}_1 \cdots \mathtt{i}_n$ over $I \cup \mathit{TO}[\mathbb{N}^{>0}]$:

- ▶ $q_0^{\mathcal{M}} \xrightarrow[u_1]{\mathtt{i}_1/o_1} q_1 \cdots \xrightarrow[u_n]{\mathtt{i}_n/o_n} q_n$ is a feasible run in $\mathcal{M}$ if and only if $q_0^{\mathit{zone}(\mathcal{M})} \xrightarrow[u'_1]{\mathtt{i}_1/o'_1} q'_1 \cdots \xrightarrow[u'_n]{\mathtt{i}_n/o'_n} q'_n$ is a feasible run in $\mathit{zone}(\mathcal{M})$.
- ▶ Moreover,
  - $o_j = o'_j$ for all $j \in \{1, \dots, n\}$, and
  - $q_j \xrightarrow{\mathtt{i}_j} \cdots \xrightarrow{\mathtt{i}_k} q_k$ is spanning $\Rightarrow u_j = (x, c) \wedge u'_j = (x', c') \wedge c = c'$.

Let $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ be a symbolic word such that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} q_n$ is a feasible run of $\mathcal{M}$. Hence, there exists $w = i_1 \cdots i_n$ such that $\overline{w} = \mathtt{w}$ and $q_0^{\mathcal{M}} \xrightarrow[u_1]{i_1/o_1} q_1 \cdots \xrightarrow[u_n]{i_n/o_n} q_n$ is a feasible run of $\mathcal{M}$. By Corollary C.8.2, it follows that $(q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow[u'_1]{i_1/o'_1} (q_1, Z_1) \cdots \xrightarrow[u'_n]{i_n/o'_n} (q_n, Z_n)$ is a feasible run of $\mathit{zone}(\mathcal{M})$. By construction of $\mathit{zone}(\mathcal{M})$, we immediately have that $o_j = o'_j$ and $u_j = u'_j$ for every j. Therefore, $\overline{i'_1 \cdots i'_n} = \mathtt{w}$ and $(q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{\mathtt{w}} (q_n, Z_n)$ is a feasible run of $\mathcal{M}$. Hence, the direction from $\mathcal{M}$ to $\mathit{zone}(\mathcal{M})$ holds. The other direction follows with the same arguments. We thus conclude that $\mathcal{M} \overset{\mathrm{sym}}{\approx} \mathit{zone}(\mathcal{M})$. □

Finally, we show that any run of $\mathit{zone}(\mathcal{M})$ is feasible.

Lemma C.8.5. *Let $\mathcal{M}$ be a sound and complete MMT. Then, any run of $\mathit{zone}(\mathcal{M})$ is feasible.*

Proof. As all states of $\mathit{zone}(\mathcal{M})$ are reachable, we can restrict the proof to runs starting at the initial state of $\mathit{zone}(\mathcal{M})$. To get Lemma C.8.5, let us prove that for any run $\pi = (q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{w} (q, Z)$ of $\mathit{zone}(\mathcal{M})$, for any $\kappa \in Z$, there exists a timed run

$$\rho = ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{v} ((q, Z), \kappa)$$

such that $\mathit{untime}(\rho) = \pi$. We prove this property by induction over $n = |w|$.
Base case: $n = 0$, *i.e.*, $w = \varepsilon$. Let $\pi = (q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{\varepsilon} (q_0^{\mathcal{M}}, \{\emptyset\})$. It is clear that there exists $\rho = ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{d} ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset)$ and $\emptyset \in \{\emptyset\}$ for any $d \in \mathbb{R}^{\geq 0}$. And we have $\mathit{untime}(\rho) = \pi$.
Induction step: Let $k \in \mathbb{N}$ and assume the proposition holds for every word of length k. Let w of length $k+1$, *i.e.*, we can decompose $w = w' \cdot i$ with $i \in A(\mathcal{M})$ and $|w'| = k$. We show that, if $\pi = (q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{w} (q, Z)$ is a run and κ is a valuation in Z, there exists a timed run $\rho = ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{v} ((q, Z), \kappa)$ such that $\mathit{untime}(\rho) = \pi$.
Assume that π is a run of $\mathit{zone}(\mathcal{M})$ and let $\pi' = (q_0^{\mathcal{M}}, \{\emptyset\}) \xrightarrow{w'} (p, Z_p)$. Observe that π' is a sub-run of π. By the induction hypothesis, we know that for any $\lambda \in Z_p$, there exists a timed run $\rho' = ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) \xrightarrow{v'} ((p, Z_p), \lambda)$ such that $\mathit{untime}(\rho') = \pi'$. Hence, let us focus on the last transition $(p, Z_p) \xrightarrow{i} (q, Z)$ of π. By construction of $\mathit{zone}(\mathcal{M})$, given (q, Z) and $\kappa \in Z$, we deduce that there exists $d \in \mathbb{R}^{\geq 0}$ and $\lambda \in Z_p$ such that

$$((p, Z_p), \lambda) \xrightarrow{i} ((q, Z), \kappa + d) \xrightarrow{d} ((q, Z), \kappa).$$

Thus, by the induction hypothesis with this λ, it follows that we have the timed run

$$\begin{aligned}\rho = ((q_0^{\mathcal{M}}, \{\emptyset\}), \emptyset) &\xrightarrow{v'} ((p, Z_p), \lambda) \xrightarrow{i} ((q, Z), \kappa + d) \\ &\xrightarrow{d} ((q, Z), \kappa)\end{aligned}$$

such that $\mathit{untime}(\rho) = \pi$. ☐

We thus have shown every piece of Theorem 9.7.5.

C.9. Proof of Proposition 10.2.4

Proposition 10.2.4. *For any s-learnable MMTs, the three symbolic queries can be implemented via a polynomial number of concrete output and equivalence queries.*

In order to prove this lemma, we first explain, given a state q, how to construct a tiw ensuring that the unique (by carefully choosing delays and leveraging the fact that the MMT is race-avoiding) timed run reading the tiw ends in q. The idea is to leverage the constraints collected on a run from the initial state to q, as introduced in Section 9.2.3. More specifically, we add constraints forcing every delay to be non-zero, in order to guarantee. Then, in Section C.9.2, we prove the proposition, query type by query type.

C.9.1. Construction of a timed run reaching a state

Let

$$\pi = p_0 \xrightarrow[u_1]{i_1/o_1} p_1 \xrightarrow[u_2]{i_2/o_2} \cdots \xrightarrow[u_n]{i_n/o_n} p_n \in \mathit{runs}(\mathcal{M})$$

with $p_0 = q_0^{\mathcal{M}}$ (*i.e.*, we start from the initial state). As $\mathcal{M}$ is s-learnable, π is feasible. We explain how to construct a tiw w such that there exists a unique run in $\mathcal{M}$ reading w and whose untimed projection is π. We do this in two steps: we first construct a timed word over $A(\mathcal{M})$ and then transform it to remove the timeout symbols.

Since π is feasible, there exists a timed run

$$\begin{aligned}\rho = (p_0, \emptyset) \xrightarrow{d_1} (p_0, \emptyset) \xrightarrow[u_1]{i_1/o_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots \\ \xrightarrow[u_n]{i_n/o_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1})\end{aligned}$$

such that $\mathit{untime}(\rho) = \pi$. Moreover, as $\mathcal{M}$ is race-avoiding, we can assume that ρ does not have any race, *i.e.*,

- ▶ $d_j > 0$ for any $j \in \{1, \dots, n+1\}$,
- ▶ $(\kappa_j - d_{j+1})(x) = 0$ if and only if $i_{j+1} = \mathit{to}[x]$ for all $j \in \{1, \dots, n-1\}$ and some $x \in \chi(p_j)$, and
- ▶ $(\kappa_n - d_{n+1})(x) \neq 0$ for all $x \in \chi(p_n)$.

We can thus refine the constraints already introduced in Section 9.2.3 in order to ensure that the timed run which can be computed from a solution to cnstr(π) satisfies the three above items.

Let $w = d_1 i_1 \cdots d_n i_n d_{n+1}$ be the timed word over $A(\mathcal{M})$ composed of the delays and actions seen along ρ. Recall that any timeout can only occur if the timer was started on a previous transition and enough time elapsed, *e.g.*, if $u_j = (x, c)$, $i_k = \mathit{to}[x]$, and x is not restarted between i_{j+1} and i_k (*i.e.*, the run from is p_{j-1} to p_k is x-spanning[10]), then the sum of the delays d_{j+1} to d_k must be equal to c. In a similar manner, if $u_j = (x, c)$ but there are no k such that $i_k = \mathit{to}[x]$, then either x was restarted, stopped or the run ended before x could reach zero. Hence, w must satisfy the following set of constraints:

10: In other words, j and k belong to the same block.

- ▶ for all $j \in \{1, \dots, n+1\}$, $d_j \in \mathbb{R}^{>0}$,
- ▶ for any j and k such that $p_{j-1} \xrightarrow[(x,c)]{i_j} p_j \xrightarrow{i_{j+1}} \cdots \xrightarrow{i_k = \mathit{to}[x]} p_k$ is an x-spanning run, the sum of the delays d_{j+1} to d_k must be equal to c, *i.e.*, $\sum_{\ell=j+1}^{k} d_\ell = c$, and
- ▶ for any j such that $u_j = (x, c)$ and there is no $k > j$ such that $i_k = \mathit{to}[x]$, then either x is restarted or stopped by some transition, or the last action i_n is read before c units of time elapse.
  - In the first case, let $k > j$ such that $i_k \neq \mathit{to}[x]$ and $p_{k-1} \xrightarrow{i_k}$ restarts or stops x. Then, the sum of the delays d_{j+1} to d_k must be strictly less than c, *i.e.*, $\sum_{\ell=j+1}^{k} d_\ell < c$.

- In the second case (so, $x \in \chi(p_n)$ and x does not time out after waiting d_{n+1}), the sum of the delays d_{j+1} to d_{n+1} must be strictly less than c, *i. e.*, $\sum_{\ell=j+1}^{n+1} d_\ell < c$.

The differences with the constraints introduced in Section 9.2.3 are that all delays must be positive, and that the sum of the delays for the last two cases is strictly smaller than c, *i. e.*, we explicitly forbid two events to occur at the same time.

As in Section 9.2.3, observe that these constraints are all linear. Moreover, if we consider the delays d_j as *variables*, one can still gather the constraints and use them to find a value for each d_j. We again denote by $\mathrm{cnstr}(\pi)$ the set of constraints for π over the variables representing the delays, and Lemma 9.2.12 still holds.

Lemma 9.2.12. Let $\mathcal{M}$ be a sound and complete MMT and $\pi \in \mathit{runs}(\mathcal{M})$. Then, $\mathrm{cnstr}(\pi)$ has a solution if and only if π is feasible.

It remains to explain how to construct a tiw w' from the timed word $w = d_1 \cdot i_1 \cdots d_n \cdot i_n \cdot d_{n+1}$ over $A(\mathcal{M})$, *i. e.*, how to drop the timeouts while still inducing the same timed run. If $i_j = \mathit{to}[x]$ for some timer x, we remove i_j and replace the delay d_j by $d_j + d_{j+1}$. We repeat this until all timeouts are removed from w. Observe that w' contains at most as many symbols as w and the sum of the delays of w' is equal to the sum of the delays of w. To simplify the rest of this section, we assume from now on that $\mathrm{cnstr}(\pi)$ provides a tiw satisfying the constraints. Furthermore, for a state q, we write $\mathrm{cnstr}(q)$ to denote a tiw returned by $\mathrm{cnstr}(\pi)$ with π a feasible run from q_0 to q (if one exists).

Now, with partial knowledge

The above construction explains how to construct such a run when $\mathcal{M}$ is known. However, during the learning process, $\mathcal{M}$ is unknown. In this section, we explain how to construct a timed run from the data structure introduced in Section 10.3, i.e., from an observation tree $\mathcal{T}$ which only holds partial knowledge on $\mathcal{M}$. This will be useful to argue that the symbolic queries can be implemented with concrete queries by the learner (*i. e.*, without having to know $\mathcal{M}$).

In other words, the constructed tiws must come from $\mathrm{cnstr}^{\mathcal{T}}(q)$ with $q \in Q^{\mathcal{T}}$. Observe that $\mathcal{T}$ is race-avoiding when $\mathcal{M}$ is race-avoiding. Moreover, due to the partial knowledge stored in $\mathcal{T}$, we may not be aware of a timer that is started in $\mathcal{M}$, *i. e.*, we may have

$$q \xrightarrow[\bot]{i} \in \mathit{runs}(\mathcal{T}) \qquad \text{and} \qquad f(q) \xrightarrow[(x,c)]{g(i)} \in \mathit{runs}(\mathcal{M}).$$

Hence, the constructed tiw from $\mathcal{T}$ may still induce multiple runs in $\mathcal{M}$. By relying on the race-avoiding aspect of both $\mathcal{T}$ and $\mathcal{M}$, we still have a way to force determinism. In short, we can change the delays to ensure that the fractional part of the sum of the delays up to an *input* is unique.[11] Then, the sum of the delays up to any timeout must have a fractional part that is equal to the fractional part of the delays up to the input that started the timer for the first time. In other words, all $\mathit{to}[x]$-transitions that are induced by an input starting x share the same fractional part.

11: As $\mathcal{M}$ is race-avoiding, we can wiggle the blocks. In practice, rational numbers are sufficient for these fractional parts and, thus, can be perfectly encoded in a computer.

Formally, let $w = d_1 i_1 \cdots i_n d_{n+1}$ be a tiw that is constructed by $\mathrm{cnstr}^{\mathcal{T}}(q)$, and $\rho \in \mathit{tiwruns}^{\mathcal{M}}(w)$ be a timed run of $\mathcal{M}$ reading w, *i.e.*,

$$\begin{aligned}\rho = (q_0, \kappa_0) &\xrightarrow{d'_1} (q_0, \kappa_0 - d'_1) \xrightarrow{i'_1} \cdots \\ &\xrightarrow{i'_m} (q_m, \kappa_m) \xrightarrow{d'_{m+1}} (q_m, \kappa_m - d'_{m+1})\end{aligned}$$

with $q_0 = q_0^{\mathcal{M}}, \kappa_0 = \emptyset$ such that timeouts are inserted when needed and delays d_i are split accordingly. Moreover, let us denote by D_j the sum of all delays from d'_1 to d'_j, and by $\mathrm{frac}(c)$ the fractional part $c - \lfloor c \rfloor$ of $c \in \mathbb{R}^{\geq 0}$. Notice that if

$$q_{j-1} \xrightarrow[(x,c)]{i'_j} \cdots \xrightarrow{i'_k = \mathit{to}[x]} q_k$$

with $i'_j \in I$ is x-spanning, then $\mathrm{frac}(D_j) = \mathrm{frac}(D_k)$ (as $c \in \mathbb{N}^{>0}$ units of time must have elapsed between the two actions i'_j and i'_k belonging to the same block). Moreover, if

$$q_{k-1} \xrightarrow[(x,c')]{\mathit{to}[x]} \cdots \xrightarrow{i'_\ell = \mathit{to}[x]} q_\ell$$

is again x-spanning, then $\mathrm{frac}(D_j) = \mathrm{frac}(D_k) = \mathrm{frac}(D_\ell)$. So, if we carefully select the delays such that every *input* i'_j induces a unique fractional part for D_j, we ensure that two actions will never happen with a zero-delay between then. Indeed, if $i_\alpha = \mathit{to}[x]$ and $i_\beta = \mathit{to}[y]$ with $x \neq y$, then $\mathrm{frac}(D_\alpha) \neq \mathrm{frac}(D_\beta)$ and some time must elapse between the two actions. Hence, w can be constructed from $\mathcal{T}$ such that $\left|\mathit{toutputs}^{\mathcal{M}}(w)\right| = 1$. From now on, we assume that $\mathrm{cnstr}^{\mathcal{T}}(q)$ provides such a word.

C.9.2. Simulating symbolic queries with concrete queries

Let us now show that every symbolic query can be implemented via a finite number of concrete queries.

Symbolic output query

Let us start with symbolic output queries. We recall the definition. For an sw (symbolic word) $\mathtt{w}$ such that $\pi = q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \in \mathit{runs}(\mathcal{M})$, $\mathbf{OQ^s}(\mathtt{w})$ returns the sequence of outputs seen along the run π. We first give the lemma we prove here.

Lemma C.9.1. *A symbolic output query on a symbolic word of length n can be implemented with at most n concrete output queries.*

Let $\mathtt{w}$ be an sw for which we ask $\mathbf{OQ^s}(\mathtt{w})$. Our goal is to construct a tiw w such that reading w in $\mathcal{M}$ yields the timed run

$$\begin{aligned}(q_0^{\mathcal{M}}, \emptyset) &\xrightarrow{d_1} (q_0^{\mathcal{M}}, \emptyset - d_1) \xrightarrow{i_1/o_1} (q_1, \kappa_1) \xrightarrow{d_2} \cdots \\ &\xrightarrow{i_n/o_n} (q_n, \kappa_n) \xrightarrow{d_{n+1}} (q_n, \kappa_n - d_{n+1})\end{aligned}$$

such that $\overline{i_1 \cdot i_2 \cdots i_n} = \mathtt{w}$. The sequence of outputs of the timed run is then the answer that must be returned by $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w})$. That is, the answer to $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w})$ is exactly $\mathbf{OQ}(w)$, with w a well-crafter tiw. Hence, our objective is to construct such a timed input word, from the knowledge stored in the observation tree $\mathcal{T}$.

Due to how symbolic queries are used during the learning algorithm, we can assume that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \in \mathit{runs}(\mathcal{M})$.[12] Furthermore, we can assume that all but the last symbol of $\mathtt{w}$ is already in the tree.[13] That is, let

$$\pi^{\mathcal{T}} = p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_{n-1}} p_{n-1} \in \mathit{runs}(\mathcal{T})$$

such that $p_0 = q_0^{\mathcal{T}}$ and $\overline{i_1 \cdots i_{n-1}}$ is equal to $\mathtt{w}$ without its last symbol. For now, we assume that, if the last symbol of w is $\mathit{to}[j]$, the corresponding update $u_j = (x, c)$ for some timer x and positive natural c.[14] This update can be learned by first performing a symbolic wait query before the symbolic output query. We can thus set i_n to be either i if the last symbol of $\mathtt{w}$ is i, or $\mathit{to}[x]$ if the last symbol of $\mathtt{w}$ is $\mathit{to}[j]$ and $u_j = (x, \cdot)$. Hence,

$$\overline{i_1 \cdots i_{n-1} \cdot i_n} = \mathtt{w}.$$

That is, we retrieve the unique run going from $q_0^{\mathcal{T}}$ to q, define the missing symbol i_n, and convert the actions into a well-formed symbolic word.

Let us now construct a tiw w such that reading w in $\mathcal{T}$ goes through the transitions reading the above word $i_1 \cdots i_{n-1} \cdot i_n$. There are two possibilities, depending on i_n:

- If $i_n \in I$, let $v = \mathrm{cnstr}^{\mathcal{T}}(p_{n-1})$ be a tiw. By construction, the last delay of v is positive (see the constraints described in Section C.9.1). Hence, let $w = v \cdot i \cdot 0$ be a tiw in which all delays (except the last) are positive.
- If $i_n = \mathit{to}[x]$ for some $x \in \chi^{\mathcal{T}}(p_{n-1})$, we have to ensure that x can timeout after reading the timed word. Let j be the index of the last transition such that $u_j = (x, c)$. We refine the constraints of $\mathrm{cnstr}^{\mathcal{T}}(p_{n-1})$ such that the sum of the delays d_{j+1} to d_n is exactly c, *i.e.* x can time out at the end of the timed run. As we assumed that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}}$ is a feasible run of $\mathcal{M}$, these new constraints have a solution. Let w be a tiw constructed from these refined constraints.

We can thus call $\mathbf{OQ}(w)$ to obtain a sequence of outputs. However, this sequence of outputs may contain *unexpected* outputs. In order to properly explain this, let us retrieve the timed runs of $\mathcal{T}$ and $\mathcal{M}$ reading the tiw w:[15]

$$\begin{aligned}\rho^{\mathcal{T}} = (p_0, \emptyset) &\xrightarrow{d_1} (p_0, \emptyset) \xrightarrow{i_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots \\ &\xrightarrow{i_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1}) \\ \rho^{\mathcal{M}} = (p'_0, \emptyset) &\xrightarrow{d'_1} (p'_0, \emptyset) \xrightarrow{i'_1} (p'_1, \kappa'_1) \xrightarrow{d'_2} \cdots \\ &\xrightarrow{i'_m} (p'_m, \kappa'_m) \xrightarrow{d'_{m+1}} (p'_m, \kappa'_m - d'_{m+1}),\end{aligned}$$

with $p_0 = q_0^{\mathcal{T}}$ and $p'_0 = q_0^{\mathcal{M}}$. We highlight that $\rho^{\mathcal{M}}$ is the unique timed run of $\mathcal{M}$ reading w (by the constraints used to construct if; see Section C.9.1), and

12: We explain in Section 10.4.1 how to ensure this.

13: If it is not the case, we can split the symbolic output query into multiple queries.

14: This may not respect the definition of an observation tree. However, we only ask a $\mathbf{OQ}^{\mathsf{s}}(\mathtt{w})$ when we want to add new nodes in the tree. Hence, the transition reading $\mathit{to}[x]$ will be added after the query and the tree will be well-formed again.

15: This means that we have to add timeout symbols when needed.

$\rho^{\mathcal{T}}$ is the unique timed run of $\mathcal{T}$ reading w.

Since $\mathcal{T}$ only holds partial knowledge about $\mathcal{M}$, it is possible that the constraints to build w are not enough, in the sense that we wait for too long in some configuration of the run in $\mathcal{M}$. Then, $\mathcal{M}$ has to process a timeout-transition that is *unexpected*, *i.e.*, there is no timeout-transition at that specific moment in the run of $\mathcal{T}$.[16] Let $i'_k = to[x']$ be the first unexpected timeout-transition. Since any $to[x']$-transition implies that x' was started before, there must exist an index $j \in \{1, \dots, k-1\}$ such that

16: An input-transition can never be unexpected.

$$(p'_{j-1}, \kappa'_{j-1}) \xrightarrow{d'_j} (p'_{j-1}, \kappa'_{j-1} - d'_j) \xrightarrow[(x',c)]{i'_j} \cdots \xrightarrow{i'_k = to[x']}$$

is x'-spanning. Moreover, as i'_k is the first such unexpected timeout, it must be that $d'_\ell = d_\ell$ for all $\ell \in \{1, \dots, k-1\}$. Finally, by construction of w, we deduce that we have

$$(p_{j-1}, \kappa_{j-1}) \xrightarrow{d_j} (p_{j-1}, \kappa_{j-1} - d_j) \xrightarrow[\bot]{i_j}$$

in $\mathcal{T}$. Indeed, otherwise, we would have an update (x, c) (with the same c by the fact that $\mathcal{T}$ is an observation tree for $\mathcal{M}$), *i.e.*, i'_k would not be unexpected, as both timed runs use the same delays up to that point.

Then, it means that we discovered a new enabled timer in p_{k-1}. Moreover, we know, by the fact that each timeout is associated with a unique fractional part, that it must be the transition reading i_j that (re)starts a timer. Let y be x_{p_j} if $i_j \in I$, and be x if $i_j = to[x]$. Then, we can create the transition $p_{k-1} \xrightarrow{to[y]}$ and change the update of the transition reading i_j to replace $\bot$ by an update (y, c).[17]

17: The constant c can be inferred from the delays in $\rho^{\mathcal{T}}$.

Hence, if an unexpected timeout occurs, we can add more nodes and transitions, and change some updates, which refines the constraints of $\mathrm{cnstr}^{\mathcal{T}}(p_n)$. After expanding the tree, we create a new tiw w as explained above, until there is no unexpected timeout, *i.e.*, until $\mathbf{OQ}(w)$ gives a "good" word, with regards to what we want to observe. Notice there can only be at most n unexpected timeouts, with n the length of the symbolic word, which proves Lemma C.9.1.

Symbolic wait query

Let us proceed with symbolic output queries. We recall the definition. For an sw (symbolic word) $\mathtt{w}$ inducing a concrete run $\pi = q_0^{\mathcal{M}} \xrightarrow{i_1} \cdots \xrightarrow{i_n} q_n \in \mathit{runs}(\mathcal{M})$ such that $\overline{i_1 \cdots i_n} = \mathtt{w}$, $\mathbf{WQ}^{\mathrm{s}}(\mathtt{w})$ returns the set of all pairs (j, c) such that $q_{j-1} \xrightarrow[(x,c)]{i_j} \cdots \xrightarrow{i_n} q_n \xrightarrow{to[x]}$ is x-spanning. We first give the lemma we prove here.

Lemma C.9.2. *A symbolic wait query (correct up to our guess of Δ) on a symbolic word of length n can be implemented with at most n^2 concrete output queries.*

Let $\mathtt{w}$ be an sw for which we want to call $\mathbf{WQ}^{\mathrm{s}}(\mathtt{w})$. Again, we can assume that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}}$ is a feasible run of $\mathcal{M}$. Due to how the queries are used during learning, we can also assume that $\pi^{\mathcal{T}} = q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}} q$ is a feasible run of $\mathcal{T}$. The idea is as follows: we consider every timer x that *may* time out in q (*i.e.*, that is started somewhere along $\pi^{\mathcal{T}}$), refine $\mathrm{cnstr}^{\mathcal{T}}(q)$ such that the block that may be using x is started *as soon as possible*, and ask a concrete output query to the teacher. We then analyze the returned sequence of outputs to determine whether x is an enabled timer of q.

Let $v = \mathrm{cnstr}^{\mathcal{T}}(q)$ be a tiw. Let us assume in the following that there is no unexpected timeout when performing a concrete output query (otherwise, we can proceed as explained above). So, we have that $(q_0^{\mathcal{M}}, \emptyset) \xrightarrow{v} (f(q), \kappa)$. Recall that each input in v is such that the fractional part of the delays up to the input is unique. Hence, it is sufficient to wait "long enough" in $(f(q), \kappa)$ to identify one potential enabled timer. For now, assume the learner knows a constant Δ that is at least as large as the largest constant appearing on any update of $\mathcal{M}$. That is, if we wait Δ units of time in a configuration and no timeout occurs, then we are sure that $\chi_0^{\mathcal{M}}(f(q)) = \emptyset$ (*i.e.*, we add Δ to the last delay of v). We discuss below how to deduce Δ during the learning process. Moreover, by the uniqueness of the fractional parts, it is easy to identify which transition (re)started the timer that times out.

So, we have to explain how to ensure that we eventually observe every enabled timer of $f(q)$. Recall that a timer must be started on a transition before q to be potentially active in q. Thus, we define a set

$$\mathit{Potential}(q) = \{x_p \mid p \text{ is an ancestor of } q\}$$

that contains every timer that *can* be enabled in q. Our idea is to check each timer one by one to determine whether it is enabled.

We select any timer x from $\mathit{Potential}(q)$ and refine the constraints of $\mathrm{cnstr}^{\mathcal{T}}(q)$ to enforce that the last delay is equal to Δ, and the input that initially starts x is triggered as *soon as possible* while still satisfying the other constraints of $\mathrm{cnstr}^{\mathcal{T}}(q)$. It may be that the resulting constraints for that x are not satisfiable, meaning by Lemma 9.2.12 that the run ending with $\mathit{to}[x]$ is not feasible and x can not be enabled in q. If there exists a solution, *i.e.*, a tiw v, we can ask $\mathbf{OQ}(w)$ to obtain a tow ω. By the uniqueness of the fractional parts, it is thus easy to check whether x times out by waiting in q. So, we can easily deduce which transition restarts x. Moreover, from the delays in ω, the constant of the update restarting x can be computed.

Lemma 9.2.12. Let $\mathcal{M}$ be a sound and complete MMT and $\pi \in \mathit{runs}(\mathcal{M})$. Then, $\mathrm{cnstr}(\pi)$ has a solution if and only if π is feasible.

We repeat this procedure for every timer in $\mathit{Potential}(q)$. Once this is done, we know the enabled timers of q. Let n be the number of states in the path from q_0 to q, *i.e.*, the length of the symbolic word provided as input to the procedure. The size of $\mathit{Potential}(q)$ is at most n, and, for every timer x in this set, we require at most n concrete output queries (due to the unexpected outputs). This proves Lemma C.9.2.

Guessing Δ. Let us quickly explain how the learner can infer Δ during the learning process. At first, Δ can be assumed to be any integer (preferably small when interacting with real-world systems). At some point, an update (x, c) may be learned by performing a wait query (or processing the counterexample of an equivalence query) with $c > \Delta$. That is, we now know that Δ is not the largest constant appearing in $\mathcal{M}$. We thus set Δ to be c. This implies that a new wait query must be performed in every explored state, in order to discover potentially missing enabled timers. That is, throughout the learning algorithm, the set of enabled timers in $\mathcal{T}$ may be an under-approximation of the set of enabled timers of the corresponding state in $\mathcal{M}$ (*cf.* seismic events in Algorithm 10.1 which require rebuilding the tree from the root). However, we will eventually learn the correct value of Δ (*cf.* Section C.15 for a bound relative to the unknown $\mathcal{M}$ on how many times seismic events and, more generally, the discovery of new timers, can occur).

Symbolic equivalence query

Finally, let us explain how to do a symbolic equivalence query using a concrete equivalence query and knowledge of the MMT. Let $\mathcal{H}$ be the sound and complete hypothesis provided to $\mathbf{EQ}^{\mathrm{s}}(\mathcal{H})$. Notice that we don't even need to use a concrete equivalence query. Indeed, we can implement a symbolic equivalence algorithm (similar to the reachability algorithm for MMTs explained in Section 9.5) which will satisfy the required specification. Below we present an alternative which does make use of a concrete equivalence query (in case one is already available).

Importantly, when using the alternative solution below, the implemented symbolic equivalence query will return **yes** if the hypothesis is timed equivalent to the hidden MMT (as opposed to when they are symbolically equivalent, *cf.* Proposition 9.4.5). If they are not timed equivalent, it will construct a counterexample of symbolic equivalence from the counterexample of timed equivalence. This is not exactly the specification of the symbolic equivalence query as in the rest of the paper, but it suffices for our algorithm Algorithm 10.1 to terminate and return an MMT that is timed equivalent to the hidden one we are trying to learn.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

Alternative solution. First, we call $\mathbf{EQ}(\mathcal{H})$. If the teacher answers **yes**, we also return **yes**. Otherwise, we need to construct a counterexample of symbolic equivalence. Let us give the lemma that we prove in this section.

Lemma C.9.3. *We need at most one concrete equivalence query to perform one symbolic equivalence query.*

Assume that the teacher answers a tiw w such that *toutputs*$^{\mathcal{M}}(w)$ differ from *toutputs*$^{\mathcal{H}}(w)$. As $\mathcal{M}$ is race-avoiding, we can assume that $|\mathit{toutputs}^{\mathcal{M}}(w)| = 1$. We thus need to construct from w an sw $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ such that

- either $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}} \in \mathit{runs}(\mathcal{H}) \Leftrightarrow q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \notin \mathit{runs}(\mathcal{M})$,
- or there exists $j \in \{1, \dots, n\}$ such that

$$q_0^{\mathcal{H}} \xrightarrow{\mathtt{i}_1 \cdots \mathtt{i}_{j-1}} q \xrightarrow[u]{\mathtt{i}_j / o} \in \mathit{runs}(\mathcal{H}),$$
$$q_0^{\mathcal{M}} \xrightarrow{\mathtt{i}_1 \cdots \mathtt{i}_{j-1}} q' \xrightarrow[u']{\mathtt{i}_j / o'} \in \mathit{runs}(\mathcal{M}),$$

and

- $o \neq o'$, or
- $u = (x, c)$ and $u' = (x', c')$ with $c \neq c'$, and $q \xrightarrow{\mathtt{i_j \cdots i_k}}$ is x-spanning for some $k \in \{j+1, \dots, n\}$.

Let

$$\rho^{\mathcal{T}} = (q_0^{\mathcal{T}}, \emptyset) \xrightarrow{d_1} (q_0^{\mathcal{T}}, \emptyset) \xrightarrow{i_1} (p_1, \kappa_1) \xrightarrow{d_2} \cdots \xrightarrow{i_n} (p_n, \kappa_n) \xrightarrow{d_{n+1}} (p_n, \kappa_n - d_{n+1})$$

be the run reading w in $\mathcal{T}$ and

$$\rho^{\mathcal{M}} = (q_0^{\mathcal{M}}, \emptyset) \xrightarrow{d_1'} (q_0^{\mathcal{M}}, \emptyset) \xrightarrow{i_1'} (p_1', \kappa_1') \xrightarrow{d_2'} \cdots \xrightarrow{i_m'} (p_m', \kappa_m') \xrightarrow{d_{m+1}'} (p_m', \kappa_m' - d_{m+1}')$$

be the run of $\mathcal{M}$ reading w. We have the following cases:

- ▶ If $n \neq m$, then there must exist a timeout in one run that has no corresponding timeout in the other run. Let $j \in \{1, \dots, n\}$ be the index of the first such timeout. Then, let $\mathtt{w} = \overline{i_1 \cdots i_j}$. It is clear that $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}} \in \mathit{runs}(\mathcal{H}) \Leftrightarrow q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \notin \mathit{runs}(\mathcal{M})$. Whether this case holds can be checked by comparing the number of outputs produced by $\mathcal{H}$ to that produced by $\mathcal{M}$ since (yet unknown) timeouts will result in extra outputs.
- ▶ If there exists an index j such that the sums of delays up to i_j and up to i_j' are different (meaning that a timeout is unexpected but does not change the length of the run — this, we can check based on the delays between observed outputs of the hypothesis and $\mathcal{M}$), we have two cases:
  - either $\overline{i_1 \cdots i_j} \neq \overline{i_1' \cdots i_j'}$, in which case we are in a scenario similar to the previous case,
  - or $\overline{i_1 \cdots i_j} = \overline{i_1' \cdots i_j'}$, meaning that there is an index $k \leq j$ such that

    $$p_{k-1} \xrightarrow[(x,c)]{i_k} \in \mathit{runs}(\mathcal{H})$$

    and

    $$p_{k-1}' \xrightarrow[(x',c')]{i_k'} \in \mathit{runs}(\mathcal{M})$$

    with $c \neq c'$, $i_j = \mathit{to}[x]$, and $i_j' = \mathit{to}[x']$. Thus, let $\mathtt{w} = \overline{i_1 \cdots i_j}$. (Notice the same symbolic counterexample works for both cases so we do not need to distinguish between them.)
- ▶ If none of the above cases holds, it must be that there exists a j such that:

  $$p_{j-1} \xrightarrow{i_j/o} \in \mathit{runs}(\mathcal{H})$$

  and

  $$p_{j-1}' \xrightarrow{i_j'/o'} \in \mathit{runs}(\mathcal{M})$$

  with $o \neq o'$. Thus, let $\mathtt{w} = \overline{i_1 \cdots i_j}$.

The construction above establishes the contrapositive of the implication from Proposition 9.4.5. We refer the interested reader to the formal proof of that result to convince themselves all cases above are exhaustive.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

Note that we only need at most one concrete equivalence query to implement the symbolic version of the query. This thus proves Lemma C.9.3.

Conclusion

In conclusion, by Lemmas C.9.1 to C.9.3, each symbolic query can be done via a polynomial number of concrete queries. Hence, we proved Proposition 10.2.4, which we repeat again.

Proposition 10.2.4. *For any s-learnable MMTs, the three symbolic queries can be implemented via a polynomial number of concrete output and equivalence queries.*

C.10. Proof of Lemma 10.3.13

Lemma 10.3.13. *Let $p_0, p_0', r_0 \in Q^{\mathcal{T}}$, $m : p_0 \leftrightarrow p_0'$ and $\mu : p_0 \leftrightarrow r_0$ be two matchings such that $\mathrm{dom}(m) \subseteq \mathrm{dom}(\mu)$. Let $w = i_1 \cdots i_n$ be a witness of the behavioral apartness $p_0 \#^m p_0'$ and $\mathit{read}^m_{p_0 \xrightarrow{w} p_n}(p_0') = p_0' \xrightarrow{w'} p_n'$. Let w^x be defined as follows:*

- *if $p_0 \#^m p_0'$ due to (constants), w^x is a word such that $p_{n-1} \xrightarrow{i_n} p_n \xrightarrow{w^x}$ is x-spanning,*
- *otherwise, $w^x = \varepsilon$.*

If $\mathit{read}^\mu_{p_0 \xrightarrow{w \cdot w^x}}(r_0) \in \mathit{runs}(\mathcal{T})$ with $r_n \in \mathcal{E}^{\mathcal{T}}$, then $p_0 \#^\mu r_0$ or $p_0' \#^{\mu \circ m^{-1}} r_0$.

Let $p_0, p_0', r_0 \in Q^{\mathcal{T}}$, and $m : p_0 \leftrightarrow p_0'$ and $\mu : p_0 \leftrightarrow r_0$ be two matchings such that $\mathrm{dom}(m) \subseteq \mathrm{dom}(\mu)$. Let $w \cdot w^x$, w', and the runs as described in the statement, $n = |w|$, and $\ell = |w \cdot w^x|$. Moreover, let v be the word labeling the run from p_0, *i.e.*, such that

$$\mathit{read}^\mu_{p_0 \xrightarrow{w \cdot w^x}}(r_0) = r_0 \xrightarrow{v} r_\ell$$

We write w_j (resp. w_j', v_j) for a symbol of $w \cdot w^x$ (resp. w', v).[18] We then have

18: So, $n = \ell$ whenever $w \vdash p_0 \#^m p_0'$ due to a condition that is not (constants), and $n < \ell$ otherwise.

$$\begin{aligned}
p_0 \xrightarrow{w_1} p_1 \xrightarrow{w_2} \cdots \xrightarrow{w_n} p_n \xrightarrow{w_{n+1}} \cdots \xrightarrow{w_\ell} p_\ell \\
\mathit{read}^m_{p_0 \xrightarrow{w} p_n}(p_0') = p_0' \xrightarrow{w_1'} p_1' \xrightarrow{w_2'} \cdots \xrightarrow{w_n'} p_n' \\
\mathit{read}^\mu_{p_0 \xrightarrow{w \cdot w^x} p_\ell}(r_0) = r_0 \xrightarrow{v_1} r_1 \xrightarrow{v_2} \cdots \xrightarrow{v_n} r_n \xrightarrow{v_{n+1}} \cdots \xrightarrow{v_\ell} r_\ell \\
\mathit{read}^{\mu \circ m^{-1}}_{p_0' \xrightarrow{w'} p_n'}(r_0) = \mathit{read}^\mu_{p_0 \xrightarrow{w} p_n}(r_0) = r_0 \xrightarrow{v_1} r_1 \xrightarrow{v_2} \cdots \xrightarrow{v_n} r_n
\end{aligned}$$

with $r_n \in \mathcal{E}^{\mathcal{T}}$ (by hypothesis). Observe that the run from p'_0 does not read w^x after p'_n.

A first possibility is that $p_0 \#^{\mu} r_0$ or $p'_0 \#^{\mu \circ m^{-1}} r_0$ due to structural apartness (with $w \cdot w^x$ or w' as witness). If this does not happen, from w being a witness of the behavioral apartness $p_0 \#^m p'_0$, we have to show that $p_0 \#^{\mu} r_0$ or $p'_0 \#^{\mu \circ m^{-1}} r_0$ for one case among (outputs), (constants), (sizes), or (enabled). We do it by a case analysis. Let $o, o', \omega \in O$ such that

$$o \neq o' \tag{outputs}$$

$$\begin{aligned} &u = (x, c) \\ \wedge\, &u' = (x', c') \\ \wedge\, &c \neq c' \end{aligned} \tag{constants}$$

$$\begin{aligned} &p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge \\ &|\chi_0(p_n)| \neq |\chi_0(p'_n)| \end{aligned} \tag{sizes}$$

$$\begin{aligned} &p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge \\ &\exists x \in \mathrm{dom}(m^{\pi}_{\pi'}) : \\ &x \in \chi_0(p_n) \\ \Leftrightarrow\, &m^{\pi}_{\pi'}(x) \notin \chi_0(p'_n) \end{aligned} \tag{enabled}$$

$$p_{n-1} \xrightarrow{w_n/o} p_n,$$
$$p'_{n-1} \xrightarrow{w'_n/o'} p'_n,$$

and

$$r_{n-1} \xrightarrow{v_n/\omega} r_n.$$

- If $o \neq o'$, then, necessarily, $\omega \neq o$ or $\omega \neq o'$ and we can apply (outputs) to obtain $p_0 \#^{\mu} r_0$ or $p'_0 \#^{\mu \circ m^{-1}} r_0$.
- If $|\chi_0(p_n)| \neq |\chi_0(p'_n)|$, then, necessarily, $|\chi_0(r_n)| \neq |\chi_0(p_n)|$ or $|\chi_0(r_n)| \neq |\chi_0(p'_n)|$. As $p_n, p'_n, r_n \in \mathcal{E}^{\mathcal{T}}$, we can apply (sizes) and get $p_0 \#^{\mu} r_0$ or $p'_0 \#^{\mu \circ m^{-1}} r_0$.
- Suppose now that $p_0 \#^m p'_0$ is due to (enabled).
  - If $x \in \mathrm{dom}(m)$ and $x \in \chi_0(p_n) \Leftrightarrow m(x) \notin \chi_0(p'_n)$, then $x \in \mathrm{dom}(\mu)$ and, necessarily, depending on whether $\mu(x) \in \chi_0(r_n)$ or $\mu(x) \notin \chi_0(r_n)$, we have either $x \in \chi_0(p_n) \Leftrightarrow \mu(x) \notin \chi_0(r_n)$ or $m(x) \in \chi_0(p'_n) \Leftrightarrow \mu(m^{-1}(m(x))) = \mu(x) \notin \chi_0(r_n)$. Hence, (enabled) applies (as $p_n, p'_n, r_n \in \mathcal{E}^{\mathcal{T}}$).
  - If $x_{p_k} \in \chi_0(p_n) \Leftrightarrow x_{p'_k} \notin \chi_0(p'_n)$ for some $k \in \{1, \dots, n\}$, we conclude with arguments similar to the previous case that (enabled) is also satisfied.
- Finally, if none of the above holds, $p_0 \#^m p'_0$ is due to (constants). We thus have

  $$p_{n-1} \xrightarrow[(x,c)]{w_n} p_n \xrightarrow{w_{n+1}} \cdots \xrightarrow{w_\ell} p_\ell$$
  $$p'_{n-1} \xrightarrow[(x',c')]{w'_n} p'_n$$
  $$r_{n-1} \xrightarrow[u]{v_n} r_n \xrightarrow{v_{n+1}} \cdots \xrightarrow{v_\ell} r_\ell$$

  with $c \neq c'$ and $w_\ell = \mathit{to}[x]$. Finally, let

  $$y = \begin{cases} \mu(x) & \text{if } x \in \mathrm{dom}(m) \\ x_{r_k} & \text{if } x = x_{p_k} \text{ for some } k \in \{1, \dots, n\} \end{cases}$$

  which means that $v_\ell = \mathit{to}[y]$, *i.e.*,

  $$v_\ell = \begin{cases} \mathit{to}[\mu(x)] & \text{if } x \in \mathrm{dom}(m) \\ \mathit{to}[x_{r_k}] & \text{if } x = x_{p_k} \text{ for some } k \in \{1, \dots, n\}. \end{cases}$$

  We argue that $u = (y, d)$ for some constant d that is distinct from either c or c'. Once we have this, (constants) applies and we obtain the desired result. We have three cases:

- If $w_n \in I$, it must be that $x = x_{p_n}$ as an input transition can only start a fresh timer in $\mathcal{T}$. Then, $v_n = w_n$ as $w_n \in I$, $w_\ell = \mathit{to}[x_{p_n}]$, and $y = x_{r_n}$. So, $v_\ell = \mathit{to}[x_{r_n}]$. Moreover, as the only transition that can start x_{r_n} for the first time is $r_{n-1} \xrightarrow[u]{v_n} r_n$, we conclude that $u = (y, d) = (x_{r_n}, d)$.
- If $w_n = \mathit{to}[x]$ with $x \in \mathsf{dom}(m)$, then $x \in \mathsf{dom}(\mu)$, $w_n = w_\ell = \mathit{to}[x]$, and $v_n = v_\ell = \mathit{to}[y] = \mathit{to}[\mu(x)]$. Assume $u = \bot$, *i.e.*, we do not restart y from r_{n-1} to r_n. In other words, y is not active in r_n. Recall that, in an observation tree, it is impossible to start again a timer that was previously active (as, for every timer z, there is a unique transition that can start z for the first time). So, y can not be active in $r_{\ell-1}$. But, then, v_ℓ can not be $\mathit{to}[y]$, which is a contradiction. Hence, $u = (y, d) = (\mu(x), d)$.
- If $w_n = \mathit{to}[x_{p_k}]$ with $k \in \{1, \dots, n-1\}$, then $w_n = w_\ell = \mathit{to}[x_{p_k}]$ and $v_n = v_\ell = \mathit{to}[x_{r_k}]$. With arguments similar to the previous case, we conclude that $u = (y, d) = (x_{r_k}, d)$.

C.11. Proof of Theorem 10.3.14

Theorem 10.3.14. *Let $\mathcal{T}$ be an observation tree for an s-learnable MMT $\mathcal{M}$ with the functional simulation $\langle f, g\rangle$, $p, p' \in Q^{\mathcal{T}}$, and $m : p \leftrightarrow p'$ be a matching. If $p \#^m p'$, then*

- ▶ *$f(p) \neq f(p')$, or*
- ▶ *there is $x \in \mathsf{dom}(m)$ such that $g(x) \neq g(m(x))$.*

Before showing Theorem 10.3.14, we first prove an intermediate result. Recall that, given two runs of $\mathcal{T}$

$$\pi = p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_n} p_n \qquad \text{and} \qquad \pi' = p'_0 \xrightarrow{i'_1} \cdots \xrightarrow{i'_n} p'_n,$$

$m^\pi_{\pi'} : \pi \leftrightarrow \pi'$ denotes the matching such that

$$m^\pi_{\pi'} = m \cup \{(x_{p_j}, x_{p'_j}) \mid j \in \{1, \dots, n\}\}$$

and $i'_j = m^\pi_{\pi'}(i_j)$ for all $j \in \{1, \dots, n\}$. Given such a matching $m^\pi_{\pi'} : \pi \leftrightarrow \pi'$, the next lemma states that if

- ▶ $f(p_0) = f(p'_0)$, *i.e.*, we start at the same state in $\mathcal{M}$, and
- ▶ $m(x) = m(g(x))$ for every $x \in \mathsf{dom}(m)$, *i.e.*, m behaves as g,

then $\langle f, g\rangle(\pi) = \langle f, g\rangle(\pi')$ and $m^\pi_{\pi'}(x) = g(m^\pi_{\pi'}(x))$ for all started $x \in \mathsf{dom}(m^\pi_{\pi'})$.[19]

19: Recall that timers that are not started along π or π' may appear in the matching, as we consider all possible pairs $(x_{p_k}, x_{p'_k})$.

Lemma C.11.1. *Let $p_0, p'_0 \in Q^{\mathcal{T}}$ and a matching $m : p_0 \leftrightarrow p'_0$ such that $f(p_0) = f(p'_0)$ and $g(x) = g(m(x))$ for all $x \in \mathsf{dom}(m)$. Moreover, let*

$w = i_1 \cdots i_n$ be a word such that

$$\pi = p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} p_n \in \mathit{runs}(\mathcal{T}), \text{ and}$$

$$\pi' = \mathit{read}^m_\pi(p'_0) = p'_0 \xrightarrow{i'_1} p'_1 \xrightarrow{i'_2} \cdots \xrightarrow{i'_n} p'_n \in \mathit{runs}(\mathcal{T}).$$

Then, $\langle f, g\rangle(\pi) = \langle f, g\rangle(\pi')$ and $g(x) = g(m^\pi_{\pi'}(x))$ for all $x \in \mathit{dom}(m^\pi_{\pi'})$ with $x \in \mathit{dom}(m)$ or $x = x_{p_k}$, $k \in \{1, \cdots, n\}$, such that x_{p_k} is started along π and x'_{p_k} is started along π'.

Proof. We prove the lemma by induction over n, the length of w.
Base case: $|w| = 0$, *i.e.*, $w = \varepsilon$. We thus have

$$\pi = p_0 \xrightarrow{\varepsilon} p_0 \qquad \text{and} \qquad \pi' = p'_0 \xrightarrow{\varepsilon} p'_0$$

which means that we have the following runs in $\mathcal{M}$

$$f(p_0) \xrightarrow{\varepsilon} f(p_0) \qquad \text{and} \qquad f(p'_0) \xrightarrow{\varepsilon} f(p'_0).$$

As $f(p_0) = f(p'_0)$, these runs of $\mathcal{M}$ are equal. Moreover, as $m^\pi_{\pi'} = m$, the second part of the lemma holds.
Induction step: Let $\ell \in \mathbb{N}$ and assume the lemma holds for length ℓ. Let $v = i_1 \cdots i_{\ell+1} = w \cdot i_{\ell+1}$ be a word of length $\ell + 1$ such that

$$p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_\ell} p_\ell \xrightarrow{i_{\ell+1}} p_{\ell+1} \in \mathit{runs}(\mathcal{T})$$

and

$$\mathit{read}^m_{p_0 \xrightarrow{w \cdot i_{\ell+1}} p_{\ell+1}}(p'_0) = p'_0 \xrightarrow{i'_1} \cdots \xrightarrow{i'_\ell} p'_n \xrightarrow{i'_{\ell+1}} p'_{\ell+1} \in \mathit{runs}(\mathcal{T}).$$

Let $\pi = p_0 \xrightarrow{w} p_\ell$ and $\pi' = \mathit{read}^m_\pi(p'_0) = p'_0 \xrightarrow{w'} p'_\ell$. By the induction hypothesis with w, it holds that

- ▶ the runs $\langle f, g\rangle(\pi)$ and $\langle f, g\rangle(\pi')$ are equal, and
- ▶ $g(x) = g(m^\pi_{\pi'}(x))$ for all started timers $x \in \mathsf{dom}(m^\pi_{\pi'})$.

It is thus sufficient to show that

- ▶ $\langle f, g\rangle(p_\ell \xrightarrow{i} p_{\ell+1}) = \langle f, g\rangle(p'_\ell \xrightarrow{i'} p'_{\ell+1})$, and
- ▶ $g(x_{p_{\ell+1}}) = g(x_{p'_{\ell+1}})$ if both $x_{p_{\ell+1}}$ and $x_{p'_{\ell+1}}$ are started, *i.e.*, if $x_{p_{\ell+1}} \in \chi(p_{\ell+1})$ and $x_{p'_{\ell+1}} \in \chi(p'_{\ell+1})$.

By definition of $\mathit{read}^m_{p_0 \xrightarrow{w \cdot i_{\ell+1}} p_{\ell+1}}(p'_0)$, we have

$$i'_{\ell+1} = \begin{cases} i_{\ell+1} & \text{if } i_{\ell+1} \in I \\ \mathit{to}[m(x)] & \text{if } i_{\ell+1} = \mathit{to}[x] \text{ with } x \in \mathsf{dom}(m) \\ \mathit{to}[x_{p'_k}] & \text{if } i_{\ell+1} = \mathit{to}[x_{p_k}] \text{ with } k \in \{1, \dots, \ell\} \end{cases}$$

We can be more precise for the last case, *i.e.*, when $i_{\ell+1} = \mathit{to}[x_{p_k}]$ with $k \in \{1, \dots, \ell\}$. As $p_\ell \xrightarrow{\mathit{to}[x_{p_k}]} \in \mathit{runs}(\mathcal{T})$, it must be that $x_{p_k} \in \chi(p_\ell)$. Hence, by definition of an observation tree, $x_{p_k} \in \chi(p_k)$. Likewise, as

$p'_\ell \xrightarrow{to[x_{p'_k}]} \in runs(\mathcal{T})$, it follows that $x_{p'_k} \in \chi(x_{p'_k})$. Hence, $g(x_{p_k})$ and $g(x_{p'_k})$ are both defined when the third case holds.
By definition of g, it holds that

$$g(i_{\ell+1}) = \begin{cases} i_{\ell+1} & \text{if } i_{\ell+1} \in I \\ to[g(x)] & \text{if } i_{\ell+1} = to[x] \text{ with } x \in \mathrm{dom}(m) \\ to[g(x_{p_k})] & \text{if } i_{\ell+1} = to[x_{p_k}] \text{ with } k \in \{1, \dots, \ell\} \end{cases}$$

and

$$g(i'_{\ell+1}) = \begin{cases} i'_{\ell+1} & \text{if } i'_{\ell+1} \in I \\ to[g(m(x))] & \text{if } i'_{\ell+1} = to[m(x)] \text{ with } x \in \mathrm{dom}(m) \\ to[g(x_{p'_k})] & \text{if } i'_{\ell+1} = to[x_{p'_k}] \text{ with } k \in \{1, \dots, \ell\}. \end{cases}$$

We have

- $i'_{\ell+1} = i_{\ell+1}$ if $i_{\ell+1} \in I$,
- $g(m(x)) = g(x)$ for all $x \in \mathrm{dom}(m)$ by the lemma statement, and
- by the induction hypothesis, $g(x_{p'_k}) = g(x_{p_k})$ for all $k \in \{1, \dots, \ell\}$ such that $x_{p_k} \in \chi(p_k)$ and $x_{p'_k} \in \chi(p'_k)$.

It follows that $g(i'_{\ell+1}) = g(i_{\ell+1})$.
As $f(p_\ell) = f(p'_\ell)$ by induction hypothesis and $g(i_{\ell+1}) = g(i'_{\ell+1})$, it holds by determinism of $\mathcal{M}$ that

$$\langle f, g\rangle(p_\ell \xrightarrow{i} p_{\ell+1}) = \langle f, g\rangle(p'_\ell \xrightarrow{i'} p'_{\ell+1}).$$

To complete the proof of the lemma, it remains to prove that if $x_{p_{\ell+1}} \in \chi(p_{\ell+1})$ and $x_{p'_{\ell+1}} \in \chi(p'_{\ell+1})$, then $g(x_{p_{\ell+1}}) = g(x_{p'_{\ell+1}})$. We have $x_{p_{\ell+1}} \in \chi(p_{\ell+1})$ (resp. $x_{p'_{\ell+1}} \in \chi(p'_{\ell+1})$) if the update of the transition $p_\ell \xrightarrow{i} p_{\ell+1}$ (resp. $p'_\ell \xrightarrow{i'} p'_{\ell+1}$) is equal to $(x_{p_{\ell+1}}, c)$ for some c (resp. $(x_{p'_{\ell+1}}, c')$ for some c'). As

$$\langle f, g\rangle(p_\ell \xrightarrow{i} p_{\ell+1}) = \langle f, g\rangle(p'_\ell \xrightarrow{i'} p'_{\ell+1})$$

and $\langle f, g\rangle$ is a functional simulation, by (FS3), we get that $g(x_{p_{\ell+1}}) = g(x_{p'_{\ell+1}})$ and $c = c'$. □

$$\forall q \xrightarrow[(x,c)]{i/o} q' : \quad f(q) \xrightarrow[(g(x),c)]{g(i)/o} f(q') \qquad \text{(FS3)}$$

We are now ready to prove Theorem 10.3.14, which we repeat again.

Theorem 10.3.14. *Let $\mathcal{T}$ be an observation tree for an s-learnable MMT $\mathcal{M}$ with the functional simulation $\langle f, g\rangle$, $p, p' \in Q^{\mathcal{T}}$, and $m : p \leftrightarrow p'$ be a matching. If $p \#^m p'$, then*

- *$f(p) \neq f(p')$, or*
- *there is $x \in \mathrm{dom}(m)$ such that $g(x) \neq g(m(x))$.*

Proof. Towards a contradiction, assume

- $w = i_1 \cdots i_n \vdash p \#^m p'$,
- $f(p) = f(p')$, and
- $\forall x \in \mathrm{dom}(m) : g(x) = g(m(x))$.

Let

$$\pi = p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_n} p_n \in \mathit{runs}(\mathcal{T})$$

and

$$\pi' = \mathit{read}^m_\pi(p'_0) = p'_0 \xrightarrow{i'_1} \cdots \xrightarrow{i'_n} p'_n \in \mathit{runs}(\mathcal{T})$$

with $p_0 = p$ and $p'_0 = p'$. Both runs exist in $\mathcal{T}$ as $w \vdash p_0 \#^m p'_0$.[20] By Lemma C.11.1, we thus have

$$\langle f, g\rangle(\pi) = \langle f, g\rangle(\pi') \tag{C.11.i}$$

$$\forall x \in \mathsf{dom}(m^\pi_{\pi'}) : x \text{ and } m^\pi_{\pi'}(x) \text{ are started} \Rightarrow g(x) = g(m^\pi_{\pi'}(x)). \tag{C.11.ii}$$

In particular, (C.11.i) holds for the last transition, *i. e.*,

$$f(p_{n-1}) \xrightarrow[u]{g(i_n)/o} f(p_n) \quad = \quad f(p'_{n-1}) \xrightarrow[u]{g(i'_n)/o} f(p'_n). \tag{C.11.iii}$$

Notice the same output and update, by determinism of $\mathcal{M}$.
First, if $w \vdash p_0 \#^m p'_0$ is structural, then there must exist a timer $x \in \mathsf{dom}(m^\pi_{\pi'})$ such that $x \not\# m^\pi_{\pi'}(x)$. By definition of the timer apartness, $x \neq m^\pi_{\pi'}(x)$ and there must exist a state q such that $x, m^\pi_{\pi'}(x) \in \chi(q)$. By (FS2), $g(x) \neq g(m^\pi_{\pi'}(x))$. Hence, we have a contradiction with (C.11.ii).
Hence, assume $w \vdash p_0 \#^m p'_0$ is behavioral. Let us study the different cases.

- Assume (outputs) holds, *i. e.*,

$$p_{n-1} \xrightarrow{i_n/o_n} p_n \qquad \text{and} \qquad p'_{n-1} \xrightarrow{i'_n/o'_n} p'_n$$

  with $o_n \neq o'_n$. By (FS3), (FS4), and (C.11.iii), we have $o_n = o$ and $o'_n = o$, which is a contradiction with (C.11.iii).
- Assume (constants) holds, *i. e.*,

$$p_{n-1} \xrightarrow[(x,c)]{i_n} p_n \qquad \text{and} \qquad p'_{n-1} \xrightarrow[(x',c')]{i'_n} p'_n$$

  with $c \neq c'$. By (FS3),

$$f(p_{n-1}) \xrightarrow[(g(x),c)]{g(i_n)} f(p_n) \qquad \text{and} \qquad f(p'_{n-1}) \xrightarrow[(g(x'),c')]{g(i'_n)} f(p'_n).$$

  By (C.11.iii), we get $(g(x), c) = (g(x'), c')$, which is a contradiction with $c \neq c'$.
- Assume (sizes) holds, *i. e.*, $p_n, p'_n \in \mathcal{E}^\mathcal{T}$ and $|\chi^\mathcal{T}_0(p_n)| \neq |\chi^\mathcal{T}_0(p'_n)|$. By the definition of explored states (Definition 10.3.6), we have

$$|\chi^\mathcal{T}_0(p_n)| = |\chi^\mathcal{M}_0(f(p_n))|$$

  and

$$|\chi^\mathcal{T}_0(p'_n)| = |\chi^\mathcal{M}_0(f(p'_n))|.$$

  It follows that

$$|\chi^\mathcal{M}_0(f(p_n))| \neq |\chi^\mathcal{M}_0(f(p'_n))|,$$

20: Moreover, they are the unique runs reading $i_1 \cdots i_n$ and $i'_1 \cdots i'_n$ from p_0 and p'_0, respectively.

$$\forall q \in Q^\mathcal{T}, x, y \in \chi^\mathcal{T}(q) : x \neq y \Rightarrow g(x) \neq g(y) \tag{FS2}$$

$$o \neq o' \tag{outputs}$$

$$\forall q \xrightarrow[(x,c)]{i/o} q' : f(q) \xrightarrow[(g(x),c)]{g(i)/o} f(q') \tag{FS3}$$

$$\forall q \xrightarrow[\perp]{i/o} q' : f(q) \xrightarrow{g(i)/o} f(q') \tag{FS4}$$

$$u = (x, c) \wedge u' = (x', c') \wedge c \neq c' \tag{constants}$$

$$p_n, p'_n \in \mathcal{E}^\mathcal{T} \wedge |\chi_0(p_n)| \neq |\chi_0(p'_n)| \tag{sizes}$$

Definition 10.3.6. Let $q \in Q^\mathcal{T}$ and π be the unique run from $q^\mathcal{T}_0$ to q in $\mathcal{T}$. We say that q is *explored* if

$$|\chi^\mathcal{T}_0(q)| = |\chi^\mathcal{M}_0(f(q))|.$$

Define $\mathcal{E}^\mathcal{T}$ as the maximal set of explored states of $\mathcal{T}$ that induces a subtree containing $q^\mathcal{T}_0$, *i. e.*, $p \in \mathcal{E}^\mathcal{T}$ for all $p \xrightarrow{i} q$ with $q \in \mathcal{E}^\mathcal{T}$.

which is in contradiction with $f(p_n) = f(p'_n)$.

▶ Assume (enabled) holds, *i.e.*, $p_n, p'_n \in \mathcal{E}^{\mathcal{T}}$ and there exists $x \in \mathrm{dom}(m^{\pi}_{\pi'})$ such that

$$x \in \chi_0^{\mathcal{T}}(p_n) \Leftrightarrow m^{\pi}_{\pi'}(x) \notin \chi_0^{\mathcal{T}}(p'_n).$$

Without loss of generality, suppose $x \in \chi_0^{\mathcal{T}}(p_n)$ and $m^{\pi}_{\pi'}(x) \notin \chi_0^{\mathcal{T}}(p'_n)$. Recall that, by (FS2) and the second part of Corollary 10.3.5, we have

$$y \in \chi_0^{\mathcal{T}}(q) \Leftrightarrow g(y) \in \chi_0^{\mathcal{M}}(f(q))$$

for all $q \in \mathcal{E}^{\mathcal{T}}$ and $y \in \mathrm{dom}(g)$ (see Definition 10.3.6). In order to leverage this, we thus need to argue that $m^{\pi}_{\pi'}(x) \in \mathrm{dom}(g)$, *i.e.*, the timer is started at some point. There are two cases:

- If $x \in \mathrm{dom}(m)$, then, by definition, $x \in \chi^{\mathcal{T}}(p_0)$ and $m^{\pi}_{\pi'}(x) = m(x) \in \chi^{\mathcal{T}}(p'_0)$. Hence,

$$g(x) \in \chi_0^{\mathcal{M}}(f(p_n))$$

and

$$g(m^{\pi}_{\pi'}(x)) \notin \chi_0^{\mathcal{M}}(f(p'_n))$$

As $g(m^{\pi}_{\pi'}(x)) = g(x)$ and $f(p_n) = f(p'_n)$, we deduce $g(x) \notin \chi_0^{\mathcal{M}}(f(p_n))$, which is a contradiction.

- If $x \notin \mathrm{dom}(m)$, then it must be that $x = x_{p_k}$ for some $k \in \{1, \dots, n\}$. Let k be the smallest such index. We can assume that $y \in \chi_0^{\mathcal{T}}(p_n) \Leftrightarrow m(y) \in \chi_0^{\mathcal{T}}(p'_n)$ for each $y \in \mathrm{dom}(m)$. That is, we have

$$x_{p_k} \in \chi_0^{\mathcal{T}}(p_n) \Leftrightarrow x_{p'_k} \notin \chi_0^{\mathcal{T}}(p'_n).$$

This means that $x_{p_j} \in \chi_0^{\mathcal{T}}(p_n) \Leftrightarrow x_{p'_j} \in \chi_0^{\mathcal{T}}(p'_n)$ for every $j \in \{1, \dots, k-1\}$. Recall that we assumed $x_{p_k} \in \chi_0^{\mathcal{T}}(p_n)$ and $x_{p'_k} \notin \chi_0^{\mathcal{T}}(p'_n)$.

By definition of an observation tree, this means that $p_n \xrightarrow{to[x_{p_k}]} p_{n+1} \in \mathit{runs}(\mathcal{T})$ for some state p_{n+1}. Moreover, as $f(p_n) = f(p'_n)$, $p_n, p'_n \in \mathcal{E}^{\mathcal{T}}$, and $|\chi_0^{\mathcal{T}}(p_n)| = |\chi_0^{\mathcal{T}}(p'_n)|$, there exist $x' \in \chi_0^{\mathcal{T}}(p'_n)$ and $y \in \chi_0^{\mathcal{M}}(f(p_n))$ such that

$$g(x') = g(x_{p_k}) = y$$
$$p'_n \xrightarrow{to[x']} p'_{n+1}$$

and

$$\langle f, g\rangle(p_n \xrightarrow{to[x_{p_k}]} p_{n+1}) = \langle f, g\rangle(p'_n \xrightarrow{to[x']} p'_{n+1}).$$

That is,

$$(f(p_n) = f(p'_n)) \xrightarrow{to[y]} (f(p_{n+1}) = f(p'_{n+1})).$$

Let $\ell \in \{k, \dots, n\}$ be the largest index such that

$$f(p_{\ell-1}) \xrightarrow[(y,c)]{g(i_\ell)} f(p_\ell),$$

$p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge$
$\exists x \in \mathrm{dom}(m^{\pi}_{\pi'}) :$
$x \in \chi_0(p_n)$
$\Leftrightarrow m^{\pi}_{\pi'}(x) \notin \chi_0(p'_n)$ (enabled)

$\forall q \in Q^{\mathcal{T}}, x, y \in \chi^{\mathcal{T}}(q) :$
$x \neq y \Rightarrow g(x) \neq g(y)$ (FS2)

Corollary 10.3.5. Let $\mathcal{T}$ be an observation tree for an s-learnable $\mathcal{M}$ with $\langle f, g\rangle$. Then, for all states $q \in Q^{\mathcal{T}}$, we have:

▶ $|\chi^{\mathcal{T}}(q)| \leq |\chi^{\mathcal{M}}(f(q))|$, and

▶ for all $x \in \chi_0^{\mathcal{T}}(q)$, $g(x) \in \chi_0^{\mathcal{M}}(f(q))$.

i. e., ℓ is the index of the last transition before $f(p_n)$ that (re)starts y. In other words, $\langle f,g\rangle(p_{\ell-1} \xrightarrow{i_\ell \cdots i_n \cdot \textit{to}[x_{p_k}]} p_{n+1})$ is y-spanning.[21] As

21: Observe that we may have $\ell = k$.

$$\begin{aligned}&\langle f,g\rangle(p_{\ell-1} \xrightarrow{i_\ell \cdots i_n \cdot \textit{to}[x_{p_k}]} p_{n+1})\\ &= \langle f,g\rangle(p'_{\ell-1} \xrightarrow{i'_\ell \cdots i'_n \cdot \textit{to}[x']} p'_{n+1}),\end{aligned}$$

$$\begin{aligned}&\langle f,g\rangle(\pi) \text{ is } y\text{-spanning} \Rightarrow\\ &\quad \exists x : \pi \text{ is } x\text{-spanning} \qquad \text{(FS5)}\\ &\quad\quad \wedge\, g(x) = y\end{aligned}$$

$$\begin{aligned}&\forall q \in Q^{\mathcal{T}}, x, y \in \chi^{\mathcal{T}}(q) :\\ &\quad x \neq y \Rightarrow g(x) \neq g(y) \qquad \text{(FS2)}\end{aligned}$$

it follows by (FS5) and (FS2) that $p'_{\ell-1} \xrightarrow{i'_\ell \cdots i'_n \cdot \textit{to}[x']} p'_{n+1}$ is x'-spanning, and thus

$$p'_{\ell-1} \xrightarrow[(x',c)]{i'_\ell} p'_\ell.$$

In order to obtain a contradiction, let us argue that $x' = x_{p'_k}$. Once we have this equality, we can deduce that $x_{p'_k} \in \chi_0^{\mathcal{T}}(p'_n)$ (as $x' \in \chi_0^{\mathcal{T}}(p'_n)$), which is a contradiction with our assumption that $x_{p'_k} \notin \chi_0^{\mathcal{T}}(p'_n)$. To do so, we start from the $g(i_\ell)$-transition of the run $f(p_{k-1}) \xrightarrow{g(i_k \cdots i_n) \cdot \textit{to}[y]} f(p_{n+1})$ and backtrack until we identify the transition that initially starts y.
When we consider $g(i_\ell)$, we have two cases:

* $g(i_\ell) \in I$, meaning that $i_\ell = i'_\ell = g(i_\ell) \in I$ and the corresponding transitions in $\mathcal{T}$ start a fresh timer. Hence, by definition of $\mathcal{T}$, it must be that

  $$p_{\ell-1} \xrightarrow[(x_{p_k},c)]{i_\ell} p_\ell$$

  for some $c \in \mathbb{N}^{>0}$. That is, $\ell = k$. Moreover,

  $$p'_{\ell-1} \xrightarrow[(x',c)]{i'_\ell} p'_\ell$$

  as the sub-run starting with that transition must be x'-spanning. Hence, $x' = x_{p'_k}$.
* $g(i_\ell) \notin I$, *i. e.*, $g(i_\ell) = \textit{to}[y]$ meaning that $i_\ell = \textit{to}[z]$ with $g(z) = y = g(x_{p_k})$. Since x_{p_k} and z are both active in p_ℓ and $g(x_{p_k}) = g(z)$, it must be that $x_{p_k} = z$ by the contrapositive of (FS2). Likewise, $i'_\ell = \textit{to}[x']$. We can thus seek a new y-spanning run that ends by the transition $\xrightarrow{\textit{to}[y]} f(p_\ell)$. Let $j \in \{1, \dots, \ell-1\}$ be the index of the first transition of this y-spanning run. Observe that $j \geq k$. Indeed, if $j < k$, then it is not possible for x_{p_k} to be enabled in p_ℓ (by definition of an observation tree). That is, $j \in \{k, \dots, \ell-1\}$.
  When considering the transitions at indices ℓ and n, and those at indices j and $\ell-1$, we have similar situations:
  - $p_{\ell-1} \xrightarrow{\textit{to}[x_{p_k}]}$ and $p_n \xrightarrow{\textit{to}[x_{p_k}]}$,
  - $p'_{\ell-1} \xrightarrow{\textit{to}[x']}$ and $p'_n \xrightarrow{\textit{to}[x']}$,
  - $(f(p_{\ell-1}) = f(p'_{\ell-1})) \xrightarrow{\textit{to}[y]}$ and $(f(p_n) = f(p'_n)) \xrightarrow{\textit{to}[y]}$, and

  - none of the updates between $f(p_\ell)$ and $f(p_n)$ restarts y, and likewise between $f(p_j)$ and $f(p_{\ell-1})$.

Hence, we can repeat the same arguments using j and $\ell - 1$, instead of ℓ and n.

That is, we can keep backtracking in $\mathcal{T}$ until we find a transition reading a symbol in I. As we argued, this transition must necessarily read i_k (so, $\ell = k$) and we conclude that $x' = x_{p_k}$.

In every case, we obtain that $x' = x_{p_k}$. As said above, this is enough to deduce a contradiction.

In every case, we obtain a contradiction. So, $f(p) \neq f(p')$ or $g(x) \neq g(m(x))$ for some $x \in \mathsf{dom}(m)$. □

C.12. Details on replaying a run

Let us first formalize the replay algorithm. Let $p_0, p_0' \in Q^{\mathcal{T}}$, $m : p_0 \leftrightarrow p_0'$ be a matching, and $w = i_1 \cdots i_n$ be a word such that $p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} p_n \in \mathit{runs}(\mathcal{T})$. We provide a function $\mathit{replay}^m_{p_0 \xrightarrow{w} p_n}(p_0')$ that extends the tree by replaying the run $p_0 \xrightarrow{w} p_n$ from p_0' as much as possible, or we discover a new apartness pair $p_0 \#^m p_0'$, or we discover a new active timer. Intuitively, we replay the run transition by transition while performing symbolic wait queries in every reached state in order to determine the enabled timers (which extends $\mathcal{E}^{\mathcal{T}}$). This may modify the number of active timers of p_0', meaning that m may become non-maximal. As we are only interested in maximal matchings, we stop early. This may also induce a new apartness pair $p_0 \#^m p_0'$, and we also stop early (notice that this may already hold without adding any state in $\mathcal{T}$). If the number of active timers of p_0' remains unchanged and no new apartness pair is discovered, we consider the next symbol i of w and try to replay it. Determining the next symbol i' to use in the run from p_0' follows the same idea as for $\mathit{read}^m_{p_0 \xrightarrow{w} p_n}(p_0')$. If $i \in I$, then $i' = i$ (recall that it is always possible to replay i as $\mathcal{M}$ is complete). If $i = \mathit{to}[x]$, we have three cases:

- ▶ $x \in \mathsf{dom}(m)$, in which case $i' = \mathit{to}[m(x)]$;
- ▶ $x = x_{p_k}$ is a fresh timer, *i.e.*, p_k appears on the run from p_0, in which case we consider the timer started on the corresponding transition from p_0': $i' = \mathit{to}[x_{p_k'}]$;
- ▶ none of the previous case holds. So, $x \in \chi^{\mathcal{T}}(p_0) \setminus \mathsf{dom}(m)$ and we can not replay i.

To avoid this last case, we consider the longest prefix v of w where each action i of v is an input or is such that $m(i)$ is defined or $i = \mathit{to}[x_{p_k}]$ for some state p_k.

Formally, assume that we already replayed $p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_{j-1}} p_{j-1}$ and obtained the run $p_0' \xrightarrow{i_1'} p_1' \xrightarrow{i_2'} \cdots \xrightarrow{i_{j-1}'} p_{j-1}'$, and we try to replay i_j from p_{j-1}'. We extend the tree with a symbolic output query when $i_j \in I$ and a symbolic wait query in every case. If the wait query leads to a discovery of new active

Algorithm C.1: Replaying a run $p_0 \xrightarrow{i_1 \cdots i_n}$ from p'_0.

```
1: if $p_0 \#^m p'_0$ then return APART
2: ▷ Longest prefix
3: $\ell \leftarrow 0$
4: while $\ell < n \land \neg(\exists x \in \chi^{\mathcal{T}}(p_0) \setminus \mathrm{dom}(m) : i_{\ell+1} = \mathit{to}[x])$ do
5:     $\ell \leftarrow \ell + 1$
6: for all $j \in \{1, \dots, \ell\}$ do                                ▷ Observe that $\ell \leq n$
7:     ▷ Extension of the tree
8:     if $i_j \in I$ then OQ$^s(p'_{j-1}, i_j)$
9:     WQ$^s(p'_{j-1})$
10:    ▷ Can we stop?
11:    if the number of active timers in $p'_0$ changed then return ACTIVE
12:    if $p_0 \#^m p'_0$ then return APART
13:    ▷ Next transition
14:    if $i_j \in I$ then $i'_j \leftarrow i_j$
15:    else if $i_j = \mathit{to}[x]$ with $x \in \mathrm{dom}(m)$ then $i'_j \leftarrow \mathit{to}[m(x)]$
16:    else if $\exists k \in \{1, \dots, j-1\} : i = \mathit{to}[x_{p_k}]$ then $i'_j \leftarrow \mathit{to}[x_{p'_k}]$
17:    Let $p'_j$ be the target state of $p'_{j-1} \xrightarrow{i'_j}$
18: if $\ell = n$ then return DONE
19: else
20:    WQ$^s(p'_\ell)$
21:    if $p_0 \#^m p'_0$ then return APART else return ACTIVE
```

timers of p'_0, we stop and return ACTIVE. If we can already deduce $p_0 \#^m p'_0$ from the replayed part, we also stop and return APART. Since $\neg(p_0 \#^m p'_0)$ and by the output and wait queries, there must exist $p_{j-1} \xrightarrow{i'_j}$ such that

- $i'_j = i_j$ if $i_j \in I$,
- $i'_j = \mathit{to}[m(x)]$ if $i_j = \mathit{to}[x]$ ($m(x)$ is well-defined by the considered prefix v of w), or
- $i'_j = \mathit{to}[x_{p'_k}]$.

Indeed, if the timeout-transition is not defined, then we have $p_0 \#^m p'_0$ by (enabled). Hence, we continue the procedure with the next symbol of w. If we completely replayed w and did not discover any new timer or apartness pair, we return DONE. Otherwise, we perform one last wait query and check whether we obtain apartness (by the following lemma, we return ACTIVE otherwise). Algorithm C.1 gives the pseudo-code.

$$p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \land \exists x \in \mathrm{dom}(m^{\pi}_{\pi'}) : x \in \chi_0(p_n) \Leftrightarrow m^{\pi}_{\pi'}(x) \notin \chi_0(p'_n) \quad \text{(enabled)}$$

We now prove Proposition 10.4.7.

Proposition 10.4.7. *Let $p_0, p'_0 \in Q^{\mathcal{T}}$, $m : p_0 \leftrightarrow p'_0$ be a maximal matching, and $\pi = p_0 \xrightarrow{w} \in \mathit{runs}(\mathcal{T})$. Then,*

- *if $\mathit{replay}^m_\pi(p'_0)$ returns* DONE*, then $\mathit{read}^m_\pi(p'_0)$ is now a run of $\mathcal{T}$.*
- *$\mathit{replay}^m_\pi(p'_0)$ returns* APART *or* ACTIVE *if $|\chi^{\mathcal{T}}(p_0)| > |\chi^{\mathcal{T}}(p'_0)|$ and w ends with $\mathit{to}[x]$ for some $x \in \chi^{\mathcal{T}}(p_0) \setminus \mathrm{dom}(m)$.*

Proof. Observe that the second item follows immediately from the first, given the fact that Algorithm C.1 processes a proper prefix of w in that case. That is, it is sufficient to show the first item.

Let $w = i_1 \cdots i_n$ and $\pi = p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_n} p_n$. Towards a contradiction, assume that $\mathit{replay}^m_\pi(p'_0) = \text{DONE}$ but $\mathit{read}^m_\pi(p'_0)$ is not a run of $\mathcal{T}$. Then, let $\ell \in \{1, \dots, n-1\}$ be the largest index such that

$$\mathit{read}^m_{p_0 \xrightarrow{i_1 \cdots i_\ell}}(p'_0) = p'_0 \xrightarrow{i'_1} p'_1 \xrightarrow{i'_2} \cdots \xrightarrow{i'_\ell} p'_\ell \in \mathit{runs}(\mathcal{T}).$$

Hence,

$$p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} \cdots \xrightarrow{i_\ell} p_\ell \xrightarrow{i_{\ell+1}} \in \mathit{runs}(\mathcal{T})$$

and

$$\mathit{read}^m_{p_0 \xrightarrow{i_1 \cdots i_\ell \cdot i_{\ell+1}}}(p'_0) = p'_0 \xrightarrow{i'_1} p'_1 \xrightarrow{i'_2} \cdots \xrightarrow{i'_\ell} p'_\ell \xrightarrow{i'_{\ell+1}} \notin \mathit{runs}(\mathcal{T}).$$

First, if $i_{\ell+1} \in I$, then we must have performed a symbolic output query in p'_ℓ (see Algorithm C.1), *i.e.*, $p'_\ell \xrightarrow{i'_{\ell+1}} \in \mathit{runs}(\mathcal{T})$. Second, if $i_{\ell+1} = \mathit{to}[x_{p_k}]$ for some $k \in \{1, \dots, \ell\}$, then we have that $p_0 \#^m p'_0$ by (enabled). Likewise when $i_{\ell+1} = \mathit{to}[x]$ with $x \in \mathsf{dom}(m)$.

$$p_n, p'_n \in \mathcal{E}^{\mathcal{T}} \wedge \exists x \in \mathsf{dom}(m^\pi_{\pi'}) : x \in \chi_0(p_n) \Leftrightarrow m^\pi_{\pi'}(x) \notin \chi_0(p'_n) \qquad \text{(enabled)}$$

So, assume $i_{\ell+1}$ is the timeout of some timer in $\chi^{\mathcal{T}}(p_0) \notin \mathsf{dom}(m)$. Since $\mathit{replay}^m_\pi(p'_0) = \text{DONE}$, we have that $\neg(p_0 \#^m p'_0)$ and we did not discover a new active timer in p'_0. Hence,

$$|\chi^{\mathcal{T}}_0(p_\ell)| = |\chi^{\mathcal{T}}_0(p'_\ell)| \tag{C.12.i}$$
$$\forall y \in \mathsf{dom}(m) : y \in \chi^{\mathcal{T}}_0(p_\ell) \Leftrightarrow m(y) \in \chi^{\mathcal{T}}_0(p'_\ell), \tag{C.12.ii}$$
$$\forall k \in \{1, \dots, \ell\} : x_{p_k} \in \chi^{\mathcal{T}}_0(p_\ell) \Leftrightarrow x_{p'_k} \in \chi^{\mathcal{T}}_0(p'_\ell), \tag{C.12.iii}$$

As m is maximal, we deduce from (C.12.ii) and (C.12.iii) that all enabled timers in p'_ℓ have their corresponding enabled timer in p_ℓ. However, x is an enabled timer in p_ℓ that does not appear among those corresponding timers as $x \notin \mathsf{dom}(m)$. This is in contradiction with (C.12.i). We thus conclude that $\mathit{replay}^m_\pi(p'_0) \neq \text{DONE}$. □

C.13. Constructing a generalized hypothesis

In this section, we introduce generalized MMTs, and show that a symbolically equivalent MMT always exists. This MMT suffers a factorial blowup, in general. We then give the construction of a generalized MMT from $\mathcal{T}$.

C.13.1. Generalized MMTs

In short, a generalized MMT is similar to an MMT, except that the update of a transition $q \xrightarrow{i} q'$ is now a function instead of a value in $(X \times \mathbb{N}^{>0}) \cup \{\bot\}$. This function dictates which timer is (re)started and how the other timers are renamed.

We adjust the definition of sound MMT to gMMT by requesting that the domain of such a function is exactly the set of active timers of q'. Moreover, its range must be the set of active timers of q or a natural constant. That is, each timer x' of q' must either come from an active timer x of q (we rename x into x'), or be (re)started with a constant. We also require that at most one timer is started per transition, as in MMTs. Finally, if $i = \mathit{to}[x]$, we forbid to rename x into x', *i. e.*, x' cannot be obtained from x: it must be the renaming of some other timer or be explicitly started by the transition. An example is given below.

Definition C.13.1 (Generalized Mealy machine with timers). A *generalized Mealy machine with timers* (*gMMT*, for short) is a tuple $\mathcal{M} = (I, O, X, Q, q_0, \chi, \delta)$ where:

- ▶ X is a finite set of timers (we assume $X \cap \mathbb{N}^{>0} = \emptyset$),
- ▶ Q is a finite set of states, with $q_0 \in Q$ the initial state,
- ▶ $\chi : Q \to \mathcal{P}(X)$ is a total function that assigns a finite set of active timers to each state, and
- ▶ $\delta : Q \times A(\mathcal{M}) \rightharpoonup Q \times O \times (X \rightharpoonup (X \cup \mathbb{N}^{>0}))$ is a partial transition function that assigns a state-output-update triple to a state-action pair. As usual, we write $q \xrightarrow[\mathfrak{r}]{i/o} q'$ if $\delta(q, i) = (q', o, \mathfrak{r})$.

We say that $\mathcal{M}$ is *sound* if it holds that

- ▶ $\chi(q_0) = \emptyset$,
- ▶ for all $q \xrightarrow[\mathfrak{r}]{} q'$, all of the following holds:
  - $\mathfrak{r}$ is injective,
  - $\mathsf{dom}(\mathfrak{r}) = \chi(q')$,
  - $\mathsf{ran}(\mathfrak{r}) \subset \chi(q) \cup \mathbb{N}^{>0}$, and
  - there is at most one $x \in \mathsf{dom}(\mathfrak{r})$ with $\mathfrak{r}(x) \in \mathbb{N}^{>0}$,

  and
- ▶ for all $q \xrightarrow[\mathfrak{r}]{\mathit{to}[x]} q'$, $x \in \chi(q)$ and $x \notin \mathsf{ran}(\mathfrak{r})$.

Observe that an MMT is in fact a gMMT where all renaming maps on transitions coincide with the identity function (except for those mapping to an integer, which are regular updates).

We now adapt the timed semantics of the model via the following rules. Again, they are similar to the rules for MMTs, except that we use $\mathfrak{r}$ to rename and start timers. Let $(q, \kappa), (q', \kappa')$ be two configurations of a sound gMMT:

- ▶ $(q, \kappa) \xrightarrow{d} (q, \kappa - d)$, if $\kappa(x) \geq d$ for every $x \in \chi(q)$,
- ▶ $(q, \kappa) \xrightarrow[\mathfrak{r}]{i/o} (q', \kappa')$, if $q \xrightarrow[\mathfrak{r}]{i/o} q' \in \mathit{runs}(\mathcal{M})$, and

$$\forall x \in \chi(q') \colon \kappa'(x) = \begin{cases} \mathfrak{r}(x) & \text{if } \mathfrak{r}(x) \in \mathbb{N}^{>0} \\ \kappa(\mathfrak{r}(x)) & \text{otherwise.} \end{cases}$$

Moreover, if $i = \mathit{to}[x]$, then $\kappa(x)$ must be 0.

We immediately obtain the definitions of enabled timers and complete gMMT. Moreover, it is clear that the definition of timed equivalence (Definition 9.4.2)

Definition 9.4.2. Two sound and complete MMTs $\mathcal{M}$ and $\mathcal{N}$ are *timed equivalent*, denoted by $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$, if and only if $\mathit{toutputs}^{\mathcal{M}}(w) = \mathit{toutputs}^{\mathcal{N}}(w)$ for all tiws w.

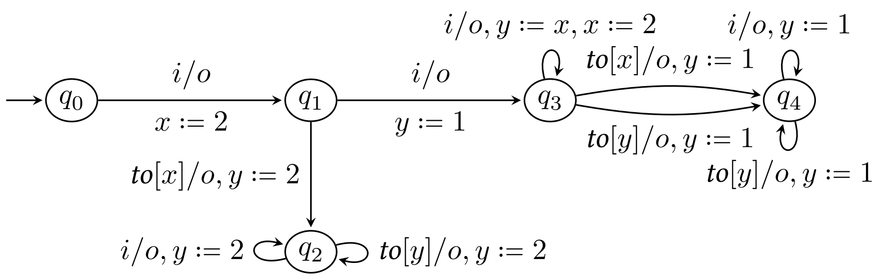


Figure C.6: A generalized MMT with $\chi(q_0) = \emptyset$, $\chi(q_1) = \{x\}$, $\chi(q_2) = \chi(q_4) = \{y\}$, and $\chi(q_3) = \{x, y\}$.

can be applied to two sound and complete gMMTs or a gMMT and an MMT, both sound and complete.

Example C.13.2. Let $\mathcal{M}$ be the gMMT of Figure C.6 with timers $X = \{x, y\}$. Update functions are shown along each transition. For instance, x is started to 2 by the transition from q_0 to q_1, while the self-loop over q_3 renames x into y (*i.e.*, y copies the current value of x) and then restarts x to 2. Let us illustrate this with the following timed run:

$$\begin{aligned}\rho = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow{i} (q_1, x = 2) \xrightarrow{0.5} (q_1, x = 1.5)\\ &\xrightarrow{i} (q_3, x = 1.5, y = 1) \xrightarrow{1} (q_3, x = 0.5, y = 0)\\ &\xrightarrow{i} (q_3, x = 2, y = 0.5) \xrightarrow{0.5} (q_3, x = 1.5, y = 0)\\ &\xrightarrow{to[y]} (q_4, y = 1) \xrightarrow{0} (q_4, y = 1).\end{aligned}$$

Observe that y takes the value of x when taking the i-loop of q_3.
We also highlight that some of the timeout transitions restart a timer that is not the one timing out, as illustrated by the timed run

$$\begin{aligned}\rho' = (q_0, \emptyset) &\xrightarrow{1} (q_0, \emptyset) \xrightarrow{i} (q_1, x = 2) \xrightarrow{2} (q_1, x = 0)\\ &\xrightarrow{to[x]} (q_2, y = 2) \xrightarrow{0} (q_2, y = 2).\end{aligned}$$

It is not hard to see that we have the following enabled timers per state

$$\begin{aligned}\chi_0(q_0) &= \emptyset & \chi_0(q_1) &= \{x\}\\ \chi_0(q_2) = \chi_0(q_4) &= \{y\} & \chi_0(q_3) &= \{x, y\}.\end{aligned}$$

From there, we conclude that $\mathcal{M}$ is complete.

Let us move towards providing a definition of symbolic equivalence (Definition 9.4.4) between a gMMT and an MMT. First, we adapt the notion of x-spanning runs. Recall that a run π of an MMT is x-spanning if

$$\pi = p_0 \xrightarrow[u_1]{i_1} p_1 \xrightarrow[u_2]{i_2} \cdots \xrightarrow[u_n]{i_n} p_n$$

with

- ▶ $u_1 = (x, c)$ for some $c \in \mathbb{N}^{>0}$,
- ▶ $u_j \neq (x, c')$ for every $j \in \{2, \ldots, n-1\}$ and $c' \in \mathbb{N}^{>0}$,

Definition 9.4.4. Two sound and complete MMTs $\mathcal{M}$ and $\mathcal{N}$ are *symbolically equivalent*, denoted by $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, if for every symbolic word $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ over A,

$$q_0^{\mathcal{M}} \xrightarrow[u_1]{\mathtt{i}_1/o_1} \cdots \xrightarrow[u_n]{\mathtt{i}_n/o_n} q_n$$

is a feasible run in $\mathcal{M}$ (where every q_j is a state of $\mathcal{M}$) if and only if

$$q_0^{\mathcal{N}} \xrightarrow[u'_1]{\mathtt{i}_1/o'_1} \cdots \xrightarrow[u'_n]{\mathtt{i}_n/o'_n} q'_n$$

is feasible in $\mathcal{N}$ (where every q'_j is a state of $\mathcal{N}$). Moreover,

- ▶ $o_j = o'_j$ for all $j \in \{1, \ldots, n\}$, and
- ▶ if $q_{j-1} \xrightarrow{\mathtt{i}_j \cdots \mathtt{i}_k} q_k$ is spanning, then $u_j = (x, c), u'_j = (x', c')$, and $c = c'$.

- $x \in \chi(p_j)$ for all $j \in \{2, \dots, n-1\}$, and
- $i_n = \mathit{to}[x]$.

The adaptation to gMMTs is straightforward: the first transition must start a timer x, each update function renames the timer (in a way, x remains active but under a different name), and the final transition reads the timeout of the timer corresponding to x. We say that a run π of a sound gMMT is *spanning* if

$$\pi = p_0 \xrightarrow[\mathfrak{r}_1]{i_1} p_1 \xrightarrow[\mathfrak{r}_2]{i_2} \cdots \xrightarrow{i_n} p_n$$

and there exist timers $x_1, \dots, x_n$ such that

- $\mathfrak{r}_1(x_1) = c$ for some $c \in \mathbb{N}^{>0}$,
- $\mathfrak{r}_j(x_j) = x_{j-1}$ for every $j \in \{2, \dots, n-1\}$ (this implies that $x_j \in \chi(p_j)$), and
- $i_n = \mathit{to}[x_{n-1}]$.

Observe that the notion of symbolic words (Section 9.4.2) still holds using this definition of spanning runs.

We can thus obtain a definition of symbolic equivalence between a sound and complete gMMT and a sound and complete MMT as follows. In short, we impose the same constraints as in Definition 9.4.4:

- a run reading a symbolic word exists in the gMMT if and only if one exists in the MMT,
- if they both exist, we must see the same outputs and for transitions starting a timer that eventually times out during the run (*i. e.*, the sub-run is spanning), we must have the same constants.

Definition C.13.3 (Symbolic equivalence between gMMT and MMT). Let $\mathcal{M}$ be a sound and complete gMMT and $\mathcal{N}$ be a sound and complete MMT. We say that $\mathcal{M}$ and $\mathcal{N}$ are *symbolically equivalent*, also denoted by $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, if for every symbolic word $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$ over $I \cup \mathit{TO}[\mathbb{N}^{>0}]$:

- $q_0^{\mathcal{M}} \xrightarrow[\mathfrak{r}_1]{\mathtt{i}_1/o_1} q_1 \cdots \xrightarrow[\mathfrak{r}_n]{\mathtt{i}_n/o_n} q_n$ is a feasible run in $\mathcal{M}$ if and only if $q_0^{\mathcal{N}} \xrightarrow[u_1']{\mathtt{i}_1/o_1'} q_1' \cdots \xrightarrow[u_n']{\mathtt{i}_n/o_n'} q_n'$ is a feasible run in $\mathcal{N}$.
- Moreover,
  - $o_j = o_j'$ for all $j \in \{1, \dots, n\}$, and
  - $q_{j-1} \xrightarrow{\mathtt{i}_j \cdots \mathtt{i}_k} q_k$ is spanning $\Rightarrow \exists x : \mathfrak{r}_j(x) = c \wedge u_j' = (x', c') \wedge c = c'$.

We then obtain that $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$ implies that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$, with arguments similar to those presented in Section C.2.

C.13.2. Existence of a symbolically equivalent Mealy machine with timers

Let $\mathcal{M}$ be a sound and complete gMMT. We give a construction of a sound and complete MMT $\mathcal{N}$ such that $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$. Intuitively, we rename the timers of $\mathcal{M}$, on the fly, into the timers of $\mathcal{N}$ and keep track in the states of $\mathcal{N}$ of the current renaming. Due to the fact that each transition of $\mathcal{M}$ can freely rename timers, it is possible that x is mapped to x_j in a state of $\mathcal{N}$, but mapped to x_k (with $k \neq j$) in some other state. That is, we sometimes need to split states of $\mathcal{M}$ into multiple states in $\mathcal{N}$, accordingly to the update functions. An example is given below.

Formally, we define $\mathcal{N} = (I, O, X^{\mathcal{N}}, Q^{\mathcal{N}}, q_0^{\mathcal{N}}, \chi^{\mathcal{N}}, \delta^{\mathcal{N}})$ with

- $X^{\mathcal{N}} = \{x_1, \dots, x_n\}$ with $n = \max_{q \in Q^{\mathcal{M}}} |\chi^{\mathcal{M}}(q)|$.
- $Q^{\mathcal{N}}$ is comprised of all (q, μ) such that
  - $q \in Q^{\mathcal{M}}$,
  - $\mu : \chi^{\mathcal{M}}(q) \leftrightarrow X^{\mathcal{N}}$, and
  - $|\mathsf{ran}(\mu)| = |\chi^{\mathcal{M}}(q)|$.

  The idea is that μ dictates how to rename a timer from $\mathcal{M}$ into a timer of $\mathcal{N}$, for this specific state. As said above, the renaming may change transition by transition.
- $q_0^{\mathcal{N}} = (q_0^{\mathcal{M}}, \emptyset)$.
- $\chi^{\mathcal{N}}((q, \mu)) = \mathsf{ran}(\mu)$ for all $(q, \mu) \in Q^{\mathcal{N}}$. Then, $\chi^{\mathcal{N}}((q, \mu))$ and $\chi^{\mathcal{M}}(q)$ have the same size.
- The function $\delta^{\mathcal{N}} : Q^{\mathcal{N}} \times A(\mathcal{N}) \rightharpoonup Q^{\mathcal{N}} \times O \times U(\mathcal{N})$ is defined as follows. Let $q \xrightarrow[\mathfrak{r}]{i/o} q'$ be a run of $\mathcal{M}$ and $(q, \mu) \in Q^{\mathcal{N}}$.
  - If $i \in I$, we have two different cases depending on whether $\mathfrak{r}$ (re)starts a fresh timer or does not (re)start anything. That is, we define
  $$\delta^{\mathcal{N}}((q, \mu), i) = ((q', \mu'), o, u)$$
  with u and μ' defined as follows.
    - If $\mathfrak{r}(x) \notin \mathbb{N}^{>0}$ for any timer x, *i.e.*, the transition does not (re)start anything, then, in $\mathcal{N}$, we also do not restart anything. Hence,
    $$u = \bot \qquad \text{and} \qquad \mu' = \mu \circ \mathfrak{r}.$$
    - If $\mathfrak{r}(x) = c \in \mathbb{N}^{>0}$ for a timer x, *i.e.*, the transition of $\mathcal{M}$ (re)starts a fresh timer, then, in $\mathcal{N}$, we want to start a timer that is not already tied to some timer. Let $\nu = \mu \circ (\mathfrak{r} \setminus \{(x, c)\})$, *i.e.*, the matching telling us how to rename every timer, except x, after taking the transition. As
    $$\begin{aligned} |X^{\mathcal{N}}| &= \max_{p \in Q^{\mathcal{M}}} |\chi^{\mathcal{M}}(q)| \\ &= \max_{(p,\nu) \in Q^{\mathcal{N}}} |\chi^{\mathcal{N}}((q, \nu))|, \end{aligned}$$
    it follows that $|X^{\mathcal{N}}| > |\mathsf{ran}(\nu)|$. Hence, there exists a timer $x_j \in X^{\mathcal{N}}$ such that $x_j \notin \mathsf{ran}(\nu)$. We then say that x is mapped to x_j and follow ν for the other timers. That is,
    $$u = (x_j, c) \qquad \text{and} \qquad \mu' = \nu \cup \{(x, x_j)\}.$$
  - If $i = \mathit{to}[x]$, we have two cases depending on whether the transi-

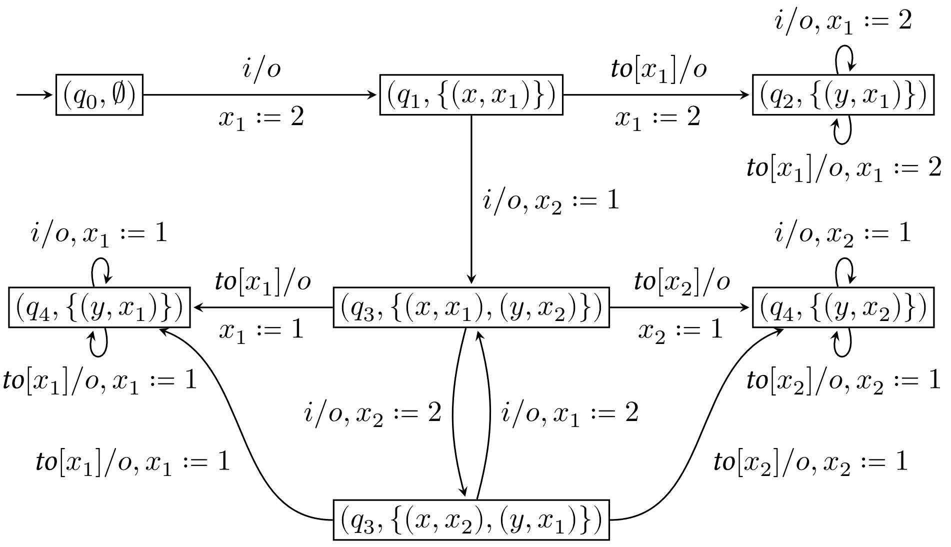


Figure C.7: The MMT obtained from the gMMT of Figure C.6.

tion start a timer, or not. That is, we define $\delta^{\mathcal{N}}((q, \mu), to[\mu(x)]) = ((q', \mu'), o, u)$ with u and μ' defined as follows.

- If $\mathfrak{r}(y) \notin \mathbb{N}^{>0}$ for any timer y, then, in $\mathcal{N}$, we do not restart anything. Hence,

$$u = \bot \qquad \text{and} \qquad \mu' = \mu \circ \mathfrak{r}.$$

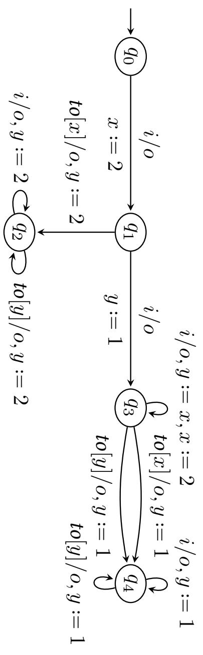
- If $\mathfrak{r}(y) = c \in \mathbb{N}^{>0}$ for some timer y, then, in $\mathcal{N}$, we want to restart x. That is, we restart the timer that times out. Again, the remaining timers simply follow $\mathfrak{r}$. Hence,

$$u = (\mu(x), c)$$

and

$$\mu' = (\mu \circ (\mathfrak{r} \setminus \{(y, c)\})) \cup \{(y, \mu(x))\}.$$

In order to obtain a deterministic procedure, let us assume that the fresh timer x_j is picked with the smallest possible j.



Example C.13.4. Let $\mathcal{M}$ be the gMMT of Figure C.6, which is repeated in the margin. Let us construct the MMT $\mathcal{N}$ according to the above procedure, using $X^{\mathcal{N}} = \{x_1, x_2\}$ as the set of timers.
We start with the initial state $(q_0, \emptyset)$ of $\mathcal{N}$. The only outgoing transition of q_0 is $q_0 \xrightarrow[(x,2)]{i/o} q_1$. That is, this transition starts a fresh timer. Hence, we define

$$(q_0, \emptyset) \xrightarrow[(x_1,2)]{i/o} (q_1, \{(x, x_1)\}).$$

Then, we consider the transition $q_1 \xrightarrow[(y,2)]{to[x]/o}$ of $\mathcal{M}$, which yields the transition

$$(q_1, \{(x, x_1)\}) \xrightarrow[(x_1,2)]{to[x_1]/o} (q_2, \{(y, x_1)\}).$$

Indeed, we use $\{(x, x_1)\}$ to know that *to*$[x]$ must become *to*$[x_1]$. As the transition in $\mathcal{M}$ starts a timer y, the transition in $\mathcal{N}$ must restart x_1 and the target state remembers that y is now mapped to x_1.
The transition $q_1 \xrightarrow[(y,1)]{i/o} q_3$ starts a fresh timer. Hence,

$$(q_1, \{(x, x_1)\}) \xrightarrow[(x_2,1)]{i/o} (q_3, \{(x, x_1), (y, x_2)\}).$$

On the contrary, the transition $q_3 \xrightarrow[(y,x),(x,2)]{i/o} q_3$ renames x into y and then restarts x to 2. So, from $(q_3, \{(x, x_1), (y, x_2)\})$ in $\mathcal{N}$, we must remember that y is now mapped to x_1, while x is now x_2, as

$$x_2 \notin \mathsf{ran}(\{(x, x_1), (y, x_2)\} \circ \{(y, x)\}) = \mathsf{ran}(\{(y, x_1)\}) = \{x_1\}.$$

That is, we define

$$(q_3, \{(x, x_1), (y, x_2)\}) \xrightarrow[(x_2,2)]{i/o} (q_3, \{(x, x_2), (y, x_1)\}).$$

In other words, the value of x_1 is left unchanged (as x is renamed into y) while x_2 is overwritten. Moreover, we indeed swap x and y in the target state. Then, using similar arguments, the same i-loop over q_3 induces an other transition that again swaps x and y:

$$(q_3, \{(x, x_2), (y, x_1)\}) \xrightarrow[(x_1,2)]{i/o} (q_3, \{(x, x_1), (y, x_2)\}).$$

This time, we restart x_1 as

$$x_1 \notin \mathsf{ran}(\{(x, x_2), (y, x_1)\} \circ \{(y, x)\}) = \mathsf{ran}(\{(y, x_2)\}) = \{x_2\}.$$

Finally, we highlight that q_4 is split into $(q_4, \{(y, x_1)\})$ and $(q_4, \{(y, x_2)\})$ due to the two timeout-transitions from q_3 to q_4 in $\mathcal{M}$.

It should be clear that $\mathcal{M}$ is sound and complete since $\mathcal{N}$ is sound and complete (and one can obtain $\mathcal{M}$ back from $\mathcal{N}$ as a sort of homomorphic image of $\mathcal{N}$).

Proposition C.13.5. *Let $\mathcal{M}$ be a sound and complete gMMT and $\mathcal{N}$ be the MMT constructed as explained above. Then, $\mathcal{N}$ is sound, complete, and its number of states is in $\mathcal{O}\left(n! \cdot \left|Q^{\mathcal{M}}\right|\right)$ with $n = \max_{q \in Q^{\mathcal{M}}} \left|\chi^{\mathcal{M}}(q)\right|$.*

The following lemma highlights the relation between a transition of $\mathcal{M}$ and a corresponding transition in $\mathcal{N}$. It holds by construction of $\mathcal{N}$.

Lemma C.13.6. *Let $(q, \mu) \in Q^{\mathcal{N}}$ and $q \xrightarrow[\mathfrak{r}]{i/o} q' \in runs(\mathcal{M})$. Then, we have the transition*

$$(q, \mu) \xrightarrow[u]{i'/o'} (q', \mu') \in runs(\mathcal{N})$$

with $o = o'$, and i' and u defined as follows.

▶ *If $i \in I$, then $i' = i$ and*

$$u = \begin{cases} (x, c) & \text{with } x \notin ran(\mu \circ \mathfrak{r}) \text{, if there exists } x \text{ such that} \\ & \mathfrak{r}(x) = c \in \mathbb{N}^{>0} \\ \bot & \text{if } \forall x : \mathfrak{r}(x) \notin \mathbb{N}^{>0}. \end{cases}$$

▶ *If $i = to[x]$, then $i' = to[\mu(x)]$ and*

$$u = \begin{cases} (\mu(x), c) & \text{if there is a timer } y \text{ such that } \mathfrak{r}(y) = c \\ \bot & \text{if } \forall x : \mathfrak{r}(x) \notin \mathbb{N}^{>0}. \end{cases}$$

Then, by applying this lemma over and over on each transition along a run, we obtain that we always see the same outputs and the same updates in both machines.

Corollary C.13.7. *For every symbolic word $\mathtt{w} = \mathtt{i}_1 \cdots \mathtt{i}_n$, we have*

$$\begin{aligned} & q_0^{\mathcal{M}} \xrightarrow[\mathfrak{r}_1]{\mathtt{i}_1/o_1} q_1 \xrightarrow[\mathfrak{r}_2]{\mathtt{i}_2/o_2} \cdots \xrightarrow[\mathfrak{r}_n]{\mathtt{i}_n/o_n} q_n \in runs(\mathcal{M}) \\ \Leftrightarrow\ & (q_0^{\mathcal{M}}, \emptyset) \xrightarrow[u_1]{\mathtt{i}_1/o_1} (q_1, \mu_1) \xrightarrow[u_2]{\mathtt{i}_2/o_2} \cdots \xrightarrow[u_n]{\mathtt{i}_n/o_n} (q_n, \mu_n) \in runs(\mathcal{N}) \end{aligned}$$

with u_j defined as follows for every j:

▶ *If $\mathtt{i}_j \in I$, then*

$$u_j = \begin{cases} (x_k, c) & \text{for some } x_k \notin ran(\mu_{j-1} \circ \mathfrak{r}_j) \text{, if there exists } x \\ & \text{such that } \mathfrak{r}_j(x) = c \in \mathbb{N}^{>0} \\ \bot & \text{if } \forall x : \mathfrak{r}_j(x) \notin \mathbb{N}^{>0}. \end{cases}$$

▶ *If $\mathtt{i}_j = to[k]$, then*

$$u_j = \begin{cases} (x, c) & \text{if there exists } x \text{ such that } \mathfrak{r}_j(x) = c \in \mathbb{N}^{>0} \text{ and} \\ & (q_{k-1}, \mu_{k-1}) \xrightarrow[(x,c')]{i_k} \\ \bot & \text{if } \forall x : \mathfrak{r}_j(x) \notin \mathbb{N}^{>0}. \end{cases}$$

This directly implies that if a run is feasible in $\mathcal{M}$, the corresponding run is also feasible in $\mathcal{N}$, and vice-versa.

Corollary C.13.8. *For any symbolic word $\mathtt{w}$, $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}} \in runs(\mathcal{M})$ is feasible if and only if $q_0^{\mathcal{N}} \xrightarrow{\mathtt{w}} \in runs(\mathcal{N})$ is feasible.*

This is enough to obtain the desired result: $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, as any run in one can be reproduced in the other, and we see the same outputs and updates (for spanning sub-runs) along these runs.

Corollary C.13.9. $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$.

C.13.3. Construction of a gMMT hypothesis

Let us now describe how a sound and complete gMMT $\mathcal{H}$ is constructed from $\mathcal{T}$. As in Section 10.4.3, we assume that the observation tree satisfies the requirements of Definition 10.4.4. Observe that for any $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$, m is necessarily bijective, for every frontier state r.

The idea is to use the set of basis states as the states of $\mathcal{H}$, with exactly the same active timers per state (that is, we do not perform any global renaming). Then,

- ▶ for any transition $q \xrightarrow[u]{i/o} q' \in \mathit{runs}(\mathcal{T})$ with $q, q' \in \mathcal{B}^{\mathcal{T}}$ and $u = (x, c)$, the update function of the transition of $\mathcal{H}$ also (re)starts x to c, and does not change the other timers, and
- ▶ for any transition $q \xrightarrow[u]{i/o} r \in \mathit{runs}(\mathcal{T})$ with $r \in \mathcal{F}^{\mathcal{T}}$, we arbitrarily select a pair $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$ and define a transition $q \xrightarrow[\mathfrak{r}]{i/o} p$ where $\mathfrak{r}$ renames every timer according to m.

In other words, the only functions that actually rename timers come from folding the tree, *i. e.*, when we exit the basis in $\mathcal{T}$.

Definition 10.4.4. In order to be able to construct a hypothesis from the observation tree, we will ensure that the following requirements are met:

Explored each basis and frontier state is explored, *i. e.*, $\mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}} \subseteq \mathcal{E}^{\mathcal{T}}$, in order to discover timers as quickly as possible,

Complete the basis is *complete*, in the sense that $p \xrightarrow{i}$ is defined for every $i \in I \cup \mathit{TO}[\chi_0^{\mathcal{T}}(p)]$, and

Active timers for every $r \in \mathcal{F}^{\mathcal{T}}$, $\mathit{compat}^{\mathcal{T}}(r) \neq \emptyset$ and $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(r)|$ for every $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$.

Definition C.13.10 (Generalized MMT hypothesis). Let $\mathcal{H} = (I, O, X^{\mathcal{H}}, Q^{\mathcal{H}}, q_0^{\mathcal{H}}, \chi^{\mathcal{H}}, \delta^{\mathcal{H}})$ be a gMMT where:

- ▶ $X^{\mathcal{H}} = \bigcup_{q \in \mathcal{B}^{\mathcal{T}}} \chi^{\mathcal{T}}(q)$,
- ▶ $Q^{\mathcal{H}} = \mathcal{B}^{\mathcal{T}}$, with $q_0^{\mathcal{H}} = q_0^{\mathcal{T}}$,
- ▶ $\chi^{\mathcal{H}}(q) = \chi^{\mathcal{T}}(q)$ for each $q \in \mathcal{B}^{\mathcal{T}}$, and
- ▶ $\delta^{\mathcal{H}}$ is constructed as follows. Let $q \xrightarrow[u]{i/o} q'$ be a transition in $\mathcal{T}$ with $q \in \mathcal{B}^{\mathcal{T}}$. We have four cases:
  - If $q' \in \mathcal{B}^{\mathcal{T}}$ (*i. e.*, the transition remains within the basis) and $u = \bot$, then we define $\delta^{\mathcal{H}}(q, i) = (q', o, \mathfrak{r})$ with
  $$\forall x \in \chi^{\mathcal{T}}(q') : \mathfrak{r}(x) = x.$$
  - If $q' \in \mathcal{B}^{\mathcal{T}}$ and $u = (y, c)$, then we define $\delta^{\mathcal{H}}(q, i) = (q', o, \mathfrak{r})$ with
  $$\forall x \in \chi^{\mathcal{T}}(q') \setminus \{y\} : \mathfrak{r}(x) = x \qquad \text{and} \qquad \mathfrak{r}(y) = c.$$
  - If $q' \in \mathcal{F}^{\mathcal{T}}$ (*i. e.*, the transition leaves the basis) and $u = \bot$, then we select an arbitrary $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$ and define $\delta^{\mathcal{H}}(q, i) = (p, o, \mathfrak{r})$ with
  $$\forall x \in \chi^{\mathcal{T}}(q') : \mathfrak{r}(m^{-1}(x)) = x.$$
  - If $q' \in \mathcal{F}^{\mathcal{T}}$ and $u = (y, c)$, then we select an arbitrary $(p, m) \in \mathit{compat}^{\mathcal{T}}(r)$ and define $\delta^{\mathcal{H}}(q, i) = (p, o, \mathfrak{r})$ with
  $$\forall x \in \chi^{\mathcal{T}}(q') : \mathfrak{r}(m^{-1}(x)) = \begin{cases} c & \text{if } x = y \\ x & \text{otherwise.} \end{cases}$$

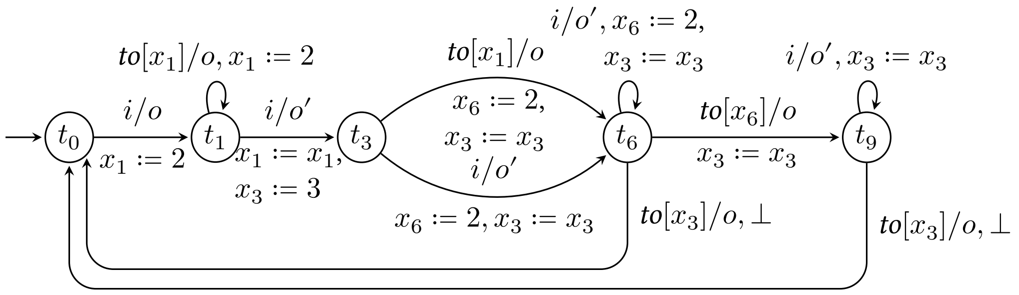
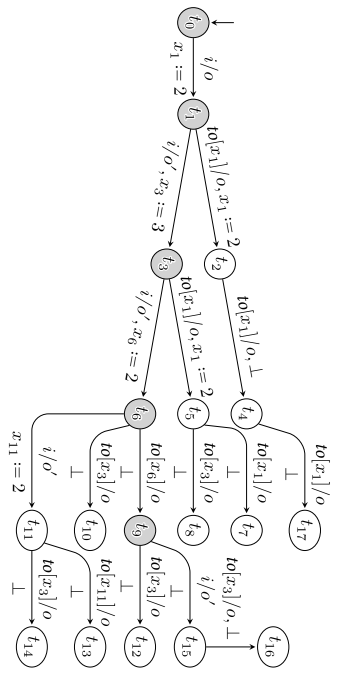


Figure C.8: A gMMT constructed from the observation tree of Figure 10.9.

It is not hard to see that $\mathcal{H}$ is sound and complete, as it is constructed from $\mathcal{T}$. Indeed, recall that $q \xrightarrow{i} \in runs(\mathcal{T})$ is defined for each $q \in \mathcal{B}^{\mathcal{T}}$ if and only if $i \in I \cup TO[\chi_0^{\mathcal{T}}(q)]$, *i.e.*, the basis is complete.



Example C.13.11. Let $\mathcal{T}$ be the observation tree of Figure 10.9. We can observe that

$$\begin{aligned}
compat^{\mathcal{T}_6}(t_2) &= \{(t_1, x_1 \mapsto x_1)\}\\
compat^{\mathcal{T}_6}(t_{10}) &= \{(t_0, \emptyset)\}\\
compat^{\mathcal{T}_6}(t_5) &= \{(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)\}\\
compat^{\mathcal{T}_6}(t_{12}) &= \{(t_0, \emptyset)\}\\
compat^{\mathcal{T}_6}(t_{11}) &= \{(t_6, x_6 \mapsto x_{11}, x_3 \mapsto x_3)\}\\
compat^{\mathcal{T}_6}(t_{15}) &= \{(t_9, x_3 \mapsto x_3)\}.
\end{aligned}$$

(See Section 10.4.6 for more details.) We thus construct a gMMT $\mathcal{H}$ as follows.

- The states of $\mathcal{H}$ are t_0, t_1, t_3, t_6, and t_9.
- The i-transition from t_0 to $t_1 \in \mathcal{B}^{\mathcal{T}}$ starts the timer x_1 to 2.
- Then, the i-transition from t_1 to $t_3 \in \mathcal{B}^{\mathcal{T}}$ keeps x_1 as-is and starts a new timer x_3.
- The i-transition from t_3 to $t_6 \in \mathcal{B}^{\mathcal{T}}$ stops x_1 and starts x_6. The timer x_3 is unchanged. Observe that, so far, we followed exactly the transitions defined within the basis of $\mathcal{T}$.
- We consider the $to[x_1]$-transition from t_3 to t_5 in $\mathcal{T}$. As $t_5 \in \mathcal{F}^{\mathcal{T}}$, we select a pair (p, m) in $compat^{\mathcal{T}}(t_5)$. Here, the only possibility is $(t_6, x_6 \mapsto x_1, x_3 \mapsto x_3)$. So, we define the renaming function $\mathfrak{r}_{5 \mapsto 6}$ such that

$$\mathfrak{r}_{5\mapsto 6}(x_6) = 2 \qquad \text{and} \qquad \mathfrak{r}_{5\mapsto 6}(x_3) = x_3.$$

  Hence, we have the transition $t_3 \xrightarrow[\mathfrak{r}_{5\mapsto 6}]{to[x_1]/o} t_6$.
- And so on.

The resulting gMMT is given in Figure C.8.
A more complex example is provided in the next section.

Finally, while one can then convert the gMMT hypothesis $\mathcal{H}$ into an MMT

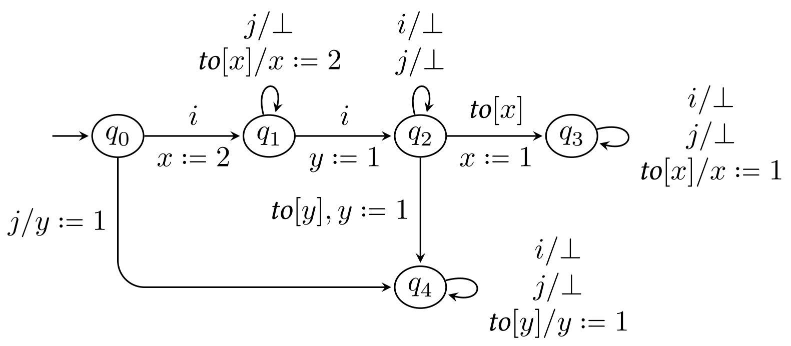


Figure C.9: The MMT $\mathcal{M}$ of Section C.13.4, with $\chi(q_0) = \emptyset, \chi(q_1) = \chi(q_3) = \{x\}, \chi(q_2) = \{x, y\}, \chi(q_4) = \{y\}$. For simplicity, the output o of each transition is omitted.

hypothesis, it is not required. Indeed, in order to avoid the factorial blowup, one can simply give $\mathcal{H}$ to the teacher who then has to check whether $\mathcal{H}$ and its hidden MMT $\mathcal{M}$ are symbolically equivalent. For instance, the teacher may construct the zone gMMT of $\mathcal{H}$ and the zone MMT of $\mathcal{M}$ and then check the equivalence between those models. This does not necessitate to construct an MMT from $\mathcal{H}$.

C.13.4. Example of a case where gMMTs are required

Finally, we give an example of an observation tree from which the construction of $\equiv$ (as explained in Section 10.4.3) fails. That is, we obtain $x \equiv y$ but $x \not\# y$. Let $\mathcal{M}$ be the MMT of Figure C.9. For simplicity, we omit all outputs in this section.

Observe that replacing the transition $q_0 \xrightarrow[(y,1)]{j} q_4$ by $q_0 \xrightarrow[(x,1)]{j} q_3$ would yield an MMT symbolically equivalent to $\mathcal{M}$. Indeed, both q_3 and q_4 have the same behavior, up to a renaming of the timer. So, the j-transition from q_0 can freely go to q_3 or q_4, under the condition that it starts respectively x or y. Here, we fix that it goes to q_4 and starts y. However, the learning algorithm may construct a hypothesis where it instead goes to q_3. In short, this uncertainty will lead us to an invalid $\equiv$.

Let $\mathcal{T}$ be the observation tree of Figure C.10. One can check that $\mathcal{T}$ is an observation tree for $\mathcal{M}$, *i. e.*, there exists a functional simulation $\langle f, g \rangle : \mathcal{T} \to \mathcal{M}$. We have the following compatible sets:

$$
\begin{aligned}
compat^{\mathcal{T}}(t_2) = compat^{\mathcal{T}}(t_{14}) &= \{(t_1, x_1 \mapsto x_1)\} \\
compat^{\mathcal{T}}(t_4) = compat^{\mathcal{T}}(t_5) = compat^{\mathcal{T}}(t_6) &= \{(t_3, x_2 \mapsto x_2)\} \\
compat^{\mathcal{T}}(t_{12}) &= \{(t_3, x_2 \mapsto x_1)\} \\
compat^{\mathcal{T}}(t_{13}) &= \{(t_3, x_2 \mapsto x_{11})\} \\
compat^{\mathcal{T}}(t_{18}) &= \{(t_{11}, x_1 \mapsto x_1, x_{11} \mapsto x_{11})\}.
\end{aligned}
$$

By constructing the equivalence relation $\equiv\ \subseteq \{x_1, x_2, x_{11}\} \times \{x_1, x_2, x_{11}\}$, we obtain that

$$x_1 \equiv x_2 \qquad \text{due to } (t_3, x_2 \mapsto x_1) \in compat^{\mathcal{T}}(t_{12})$$

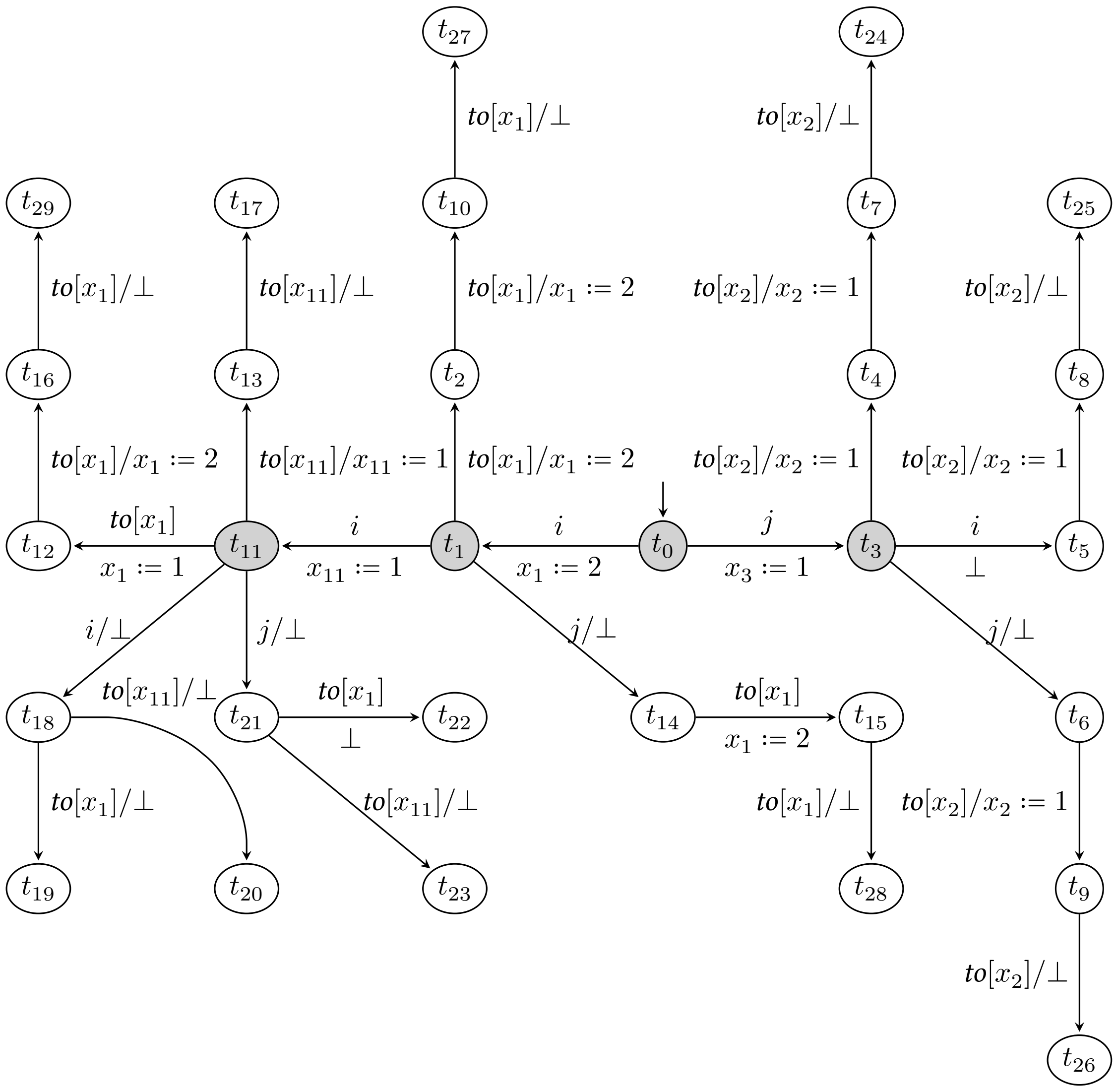


Figure C.10: The observation tree $\mathcal{T}$ of Section C.13.4. Basis states are highlighted in gray. Outputs are omitted.

and

$$x_2 \equiv x_{11} \qquad \text{due to } (t_3, x_2 \mapsto x_{11}) \in \mathit{compat}^{\mathcal{T}}(t_{13}).$$

So, $x_1 \equiv x_{11}$. However, notice that $x_1 \# x_{11}$, as both timers are active in t_{11}. Since that relation is the only possibility, we conclude that it is not always possible to construct a relation that does not put together two apart timers. Finally, Figure C.11 gives the gMMT that is constructed from $\mathcal{T}$ (observe the renaming that occurs when going from t_{11} to t_3), while Figure C.12 gives the MMT constructed from that gMMT. Notice that t_3 is split into two states, like in Figure C.9.

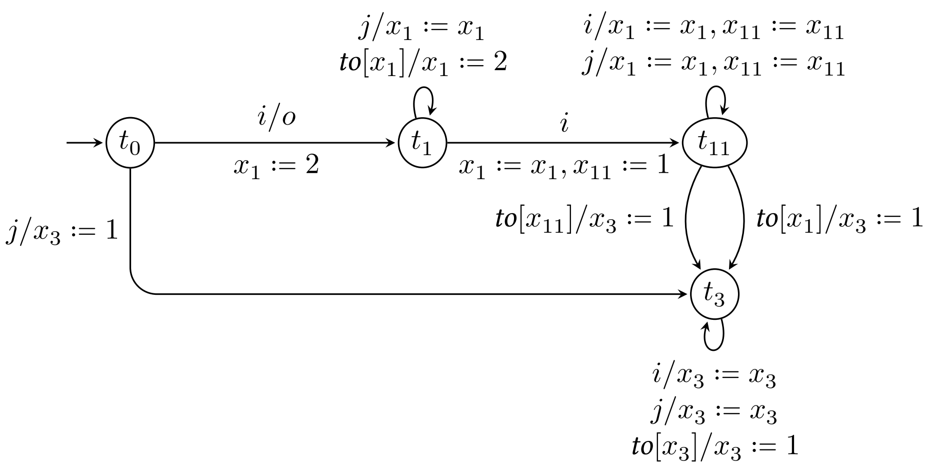


Figure C.11: The gMMT constructed from the observation tree of Figure C.10.

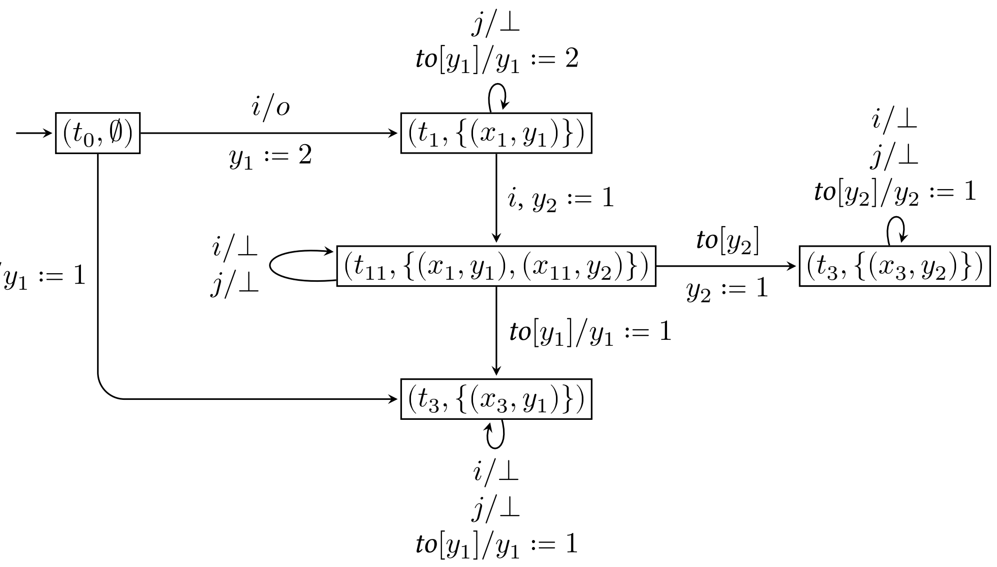


Figure C.12: The MMT constructed from the gMMT of Figure C.11.

C.14. Proof of Proposition 10.4.11

Proposition 10.4.11. *When processing a counterexample, we eventually find a j such that replay*$^{m_j^{-1}}_{v'_j}$ $\xrightarrow{}_{r_j}$ (p_j) *returns* APART *or* ACTIVE.

Towards a contradiction, assume that each call to the replay algorithm returns DONE. Let $\mathtt{w}' \cdot \mathtt{i}$ be the prefix of the counterexample as computed in Section 10.4.5. Let us assume that one of the following cases holds:[22]

22: As, otherwise, it is not necessary to apply the procedure described in Section 10.4.5.

1. The symbolic word $\mathtt{w}' \cdot \mathtt{i}$ can be read in $\mathcal{T}$ but not in $\mathcal{H}$, or vice-versa: $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}} \in \mathit{runs}(\mathcal{T})$ if and only if $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}} \notin \mathit{runs}(\mathcal{H})$. As $\mathcal{H}$ is complete and we performed a symbolic output query in $\mathcal{T}$, $\mathtt{i}$ cannot be an input. That is, $\mathtt{i} = \mathit{to}[j]$ for some index j.

2. The symbolic word $\mathtt{w}' \cdot \mathtt{i}$ can be read in both $\mathcal{T}$ and $\mathcal{H}$ but $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}'} q \xrightarrow[u]{\mathtt{i}/o}$, $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}'} q' \xrightarrow[u']{\mathtt{i}/o'}$, and $o \neq o'$ or $u = (x, c)$ and $u' = (x', c')$ with $c \neq c'$.

Let $r_1, \dots, r_n \in \mathcal{F}^{\mathcal{T}}$, and, for every j, (p_j, m_j) be the selected pair to fold r_j when building $\mathcal{H}$. Moreover, let the words $v_1, v_1', \dots, v_n, v_n', v_{n+1}$ be as constructed in Section 10.4.5, *i.e.*,

- $q_0^{\mathcal{T}} \xrightarrow{v_1} r_1 \xrightarrow{v_1'}$,
- $\forall j \in \{1, \dots, n-1\} : \mathit{read}^{m_j^{-1}}_{r_j \xrightarrow{v_j'}}(p_j) = p_j \xrightarrow{v_{j+1}} r_{j+1} \xrightarrow{v_{j+1}'}$, and
- $\mathit{read}^{m_n^{-1}}_{r_n \xrightarrow{v_n'}}(p_n) = p_n \xrightarrow{v_{n+1}} p_{n+1}$ with $p_{n+1} \in \mathcal{B}^{\mathcal{T}} \cup \mathcal{F}^{\mathcal{T}}$.

As $\mathit{replay}^{m_j^{-1}}_{r_j \xrightarrow{v_j'}}(p_j)$ returned DONE for every j, it must be that $\neg(p_j \#^{m_j} r_j)$ and no new active timer was found in p_j. So, $(p_j, m_j) \in \mathit{compat}^{\mathcal{T}}(r_j)$.

Let us study every case of the counterexample definition (see Definition 10.2.3).

- If we have the run $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}'} q \xrightarrow{\mathtt{i}/o}$ in $\mathcal{T}$ and the run $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}'} q' \xrightarrow{\mathtt{i}/o'}$ in $\mathcal{H}$ with $o \neq o'$, we are then in the case 2 above. Since $\mathcal{T}$ is an observation tree for $\mathcal{M}$, it must be that

  $$q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}'} f(q) \xrightarrow{\mathtt{i}/o} \in \mathit{runs}(\mathcal{M}).$$

  Moreover, we have

  $$q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}'} q \xrightarrow{\mathtt{i}} = q_0^{\mathcal{T}} \xrightarrow{v_1} r_1 \xrightarrow{v_1'}$$

  and

  $$\mathit{read}^{m_1^{-1}}_{r_1 \xrightarrow{v_1'}}(p_1) = p_1 \xrightarrow{v_2} r_2 \xrightarrow{v_2'}.$$

  Hence, the last output of $r_1 \xrightarrow{v_1'}$ is o. As $\neg(p_1 \#^{m_1^{-1}} r_1)$, it follows that the last output of $r_2 \xrightarrow{v_2'}$ is also o. By applying the same arguments, the last output of $r_3 \xrightarrow{v_3'}$ must be o, and so on. We thus conclude that the last output of $p_n \xrightarrow{v_{n+1}}$ is o. Observe that, by construction of $\mathcal{H}$, it must be that the end of the run $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}}$ is the run $p_n \xrightarrow{v_{n+1}}$. Therefore, we have $q' \xrightarrow{\mathtt{i}/o} \in \mathit{runs}(\mathcal{H})$, meaning that $o = o'$ (by determinism of $\mathcal{H}$), which is a contradiction.
- If we have the run $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}'} q \xrightarrow[(x,c)]{\mathtt{i}}$ in $\mathcal{T}$ and the run $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}'} q' \xrightarrow[(x',c')]{\mathtt{i}}$ in $\mathcal{H}$ with $c \neq c'$, we are then in the case 2 above. Since $\mathcal{T}$ is an observation tree for $\mathcal{M}$, it must be that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}'} f(q) \xrightarrow[(g(x),c)]{\mathtt{i}}$ is a run of $\mathcal{M}$. By applying the same arguments as in the previous case, we deduce that $q' \xrightarrow[(x',c)]{\mathtt{i}} \in \mathit{runs}(\mathcal{H})$, meaning that $c = c'$, which is a contradiction.

- If we have the run $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}}$ in $\mathcal{T}$ but $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}}$ is a not a run of $\mathcal{H}$, it must be that $\mathtt{i} = \mathit{to}[j]$ for some $j \in \{1, \dots, |\mathtt{w}|\}$. Since $\mathcal{T}$ is an observation tree for $\mathcal{M}$, (and by the part of Section 10.4.5 seeking a prefix of the provided counterexample), it must be that $q_0^{\mathcal{M}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}} \in \mathit{runs}(\mathcal{M})$. We are then in the case 1 above. By applying the same arguments as in the previous two cases, we conclude that $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}} \in \mathit{runs}(\mathcal{H})$, which is a contradiction.
- The case where $q_0^{\mathcal{T}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}}$ is not a run of $\mathcal{T}$ and $q_0^{\mathcal{H}} \xrightarrow{\mathtt{w}' \cdot \mathtt{i}}$ is a run of $\mathcal{H}$ induces a contradiction with similar arguments.

In every case, we obtain a contradiction and we conclude that there must exist $j \in \{1, \dots, n\}$ such that $p_j \#^{m_j} r_j$ or a new active timer is discovered in p_j.

C.15. Proof of Theorem 10.4.1

Theorem 10.4.1. *Let $\mathcal{M}$ be an s-learnable MMT and ζ be the length of the longest counterexample. Then,*

- *the $L^{\#}_{\mathit{MMT}}$ algorithm eventually terminates and returns an MMT $\mathcal{N}$ such that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ and whose size is polynomial in $|Q^{\mathcal{M}}|$ and factorial in $|X^{\mathcal{M}}|$, and*
- *in time and number of* $\mathbf{OQ}^{\mathrm{s}}$, $\mathbf{WQ}^{\mathrm{s}}$, $\mathbf{EQ}^{\mathrm{s}}$ *polynomial in $|Q^{\mathcal{M}}|$, $|I|$, and ζ, and factorial in $|X^{\mathcal{M}}|$.*

Before showing Theorem 10.4.1, we prove several intermediate results. First, we argue that the refinement loop of Algorithm 10.1 eventually terminates, *i. e.*, a hypothesis is eventually constructed. Under the assumption that the basis is finite, observe that the frontier is then finite (as $I \cup \mathit{TO}[\cup_{p \in \mathcal{B}^{\mathcal{T}}} \chi^{\mathcal{T}}(p)]$ is finite). Hence, we can apply **Completion**, **Active timers**, **WCT** only a finite number of times. It is thus sufficient to show that the basis cannot grow forever, *i. e.*, that **Promotion** is applied a finite number of times, which implies that **Seismic** is also applied a finite number of times. The next lemma states an upper bound over the number of basis states.

Lemma C.15.1. $|\mathcal{B}^{\mathcal{T}}| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|}$.

Proof. In order to distinguish the theoretical definition of $\mathcal{B}^{\mathcal{T}}$ and its computation, let us denote by B the basis as computed in the refinement loop of Algorithm 10.1. We highlight that B may not always satisfy the definition of $\mathcal{B}^{\mathcal{T}}$ *during* the refinement loop. Indeed, as explained in Section 10.4.4, when a new active timer is found in a basis state, we may have $\neg(p \#^m p')$ for some $p \neq p' \in \mathcal{B}^{\mathcal{T}}$ and maximal matching $m : p \leftrightarrow p'$. We thus need to perform **Seismic** and recompute B. Below, we will establish that $B \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|}$. The same (or even simpler) arguments yield the bound for $\mathcal{B}^{\mathcal{T}}$.
Let us thus assume we already treated the pending **Seismic** (if there is one), *i. e.*, that

$$\forall p, p' \in B, \text{ maximal matchings } m : p \leftrightarrow p' : p \neq p' \Rightarrow p \#^m p'. \quad \text{(C.15.i)}$$

Towards a contradiction, assume $|B| > |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|}$. By Pigeonhole principle, there must exist two states $p \neq p' \in B$ such that

- $f(p) = f(p')$, as $|B| > |Q^{\mathcal{M}}|$, and
- $g(\chi^{\mathcal{T}}(p)) = g(\chi^{\mathcal{T}}(p'))$.

This implies that $|\chi^{\mathcal{T}}(p)| = |\chi^{\mathcal{T}}(p')|$. Let $m_g : p \leftrightarrow p'$ be the matching such that $g(x) = g(m_g(x))$ for all $x \in \mathrm{dom}(m_g)$.[23] Observe that m_g is maximal. Hence, we have $p \#^{m_g} p'$ by (C.15.i). However, by the contrapositive of Theorem 10.3.14, it follows that $\neg(p \#^{m_g} p')$. We thus obtain a contradiction, meaning that both p and p' cannot be in B at the same time. Hence, $|B| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|}$. ☐

23: It necessarily exists as $g(\chi^{\mathcal{T}}(p)) = g(\chi^{\mathcal{T}}(p'))$.

Theorem 10.3.14. Let $\mathcal{T}$ be an observation tree for an s-learnable MMT $\mathcal{M}$ with the functional simulation $\langle f, g\rangle$, $p, p' \in Q^{\mathcal{T}}$, and $m : p \leftrightarrow p'$ be a matching. If $p \#^m p'$, then

- $f(p) \neq f(p')$, or
- there is $x \in \mathrm{dom}(m)$ such that $g(x) \neq g(m(x))$.

From the lemma, we immediately get bounds on the size of the frontier and the compatible sets. For the former, note that the number of immediate successors of each state is bounded by $|I| + |X^{\mathcal{M}}| = |A(\mathcal{M})|$; for the latter, the number of maximal matchings is at most $|X^{\mathcal{M}}|!$ since the number of active timers in any state from the observation tree is bounded by the same value from the hidden MMT $\mathcal{M}$.

Corollary C.15.2.

$$|\mathcal{F}^{\mathcal{T}}| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|} \cdot (|A(\mathcal{M})|)$$
$$\max_{r \in \mathcal{F}^{\mathcal{T}}} |\mathit{compat}^{\mathcal{T}}(r)| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|} \cdot |X^{\mathcal{M}}|!\ .$$

Finally, we give an upper bound over the length of the minimal words ending in $\mathit{to}[x]$ for any $q \in Q^{\mathcal{T}}$ and $x \in \chi^{\mathcal{T}}(q)$. That is, when applying **WCT**, one can seek a short run to be replayed (typically, via a BFS).

Lemma C.15.3.

$$\max_{q \in Q^{\mathcal{T}}} \max_{x \in \chi^{\mathcal{T}}(q)} \min_{q \xrightarrow{w \cdot \mathit{to}[x]} \in \mathit{runs}(\mathcal{T})} |w \cdot \mathit{to}[x]| \leq |Q^{\mathcal{M}}|$$

Proof. Towards a contradiction, assume that

$$\max_{q \in Q^{\mathcal{T}}} \max_{x \in \chi^{\mathcal{T}}(q)} \min_{q \xrightarrow{w \cdot \mathit{to}[x]} \in \mathit{runs}(\mathcal{T})} |w \cdot \mathit{to}[x]| > |Q^{\mathcal{M}}|.$$

Then, there must exist a state p_0 and a timer $x \in \chi^{\mathcal{T}}(p_0)$ such that the length of $w \cdot \mathit{to}[x]$ is strictly greater than the number of states of $\mathcal{M}$. That is, we have a run

$$\pi = p_0 \xrightarrow{i_1} \cdots \xrightarrow{i_\ell} p_\ell \xrightarrow{\mathit{to}[x]} \in \mathit{runs}(\mathcal{T})$$

with $\ell > |Q^{\mathcal{M}}|$. Observe that $p_\ell \in \mathcal{E}^{\mathcal{T}}$, as $p_\ell \xrightarrow{\mathit{to}[x]}$ is defined. By Pigeonhole principle, there must exist a $j \in \{1, \dots, \ell - 1\}$ such that $f(p_j) = f(p_\ell)$. As $\mathcal{E}^{\mathcal{T}}$ is tree-shaped, it follows that $p_j \in \mathcal{E}^{\mathcal{T}}$ (since $j < \ell$).

We thus need to argue that $p_j \xrightarrow{\mathit{to}[x]} \in \mathit{runs}(\mathcal{T})$ to obtain our contradiction.

Since $f(p_j) = f(p_\ell)$, it directly follows that

$$\chi_0^{\mathcal{M}}(f(p_j)) = \chi_0^{\mathcal{M}}(f(p_\ell)).$$

Moreover, $x \in \chi^{\mathcal{T}}(f(p_j))$, as x is active in both p_0 and p_ℓ (and $j < \ell$). Hence, $g(x) \in \chi_0^{\mathcal{M}}(f(p_j))$ as $g(x) \in \chi_0^{\mathcal{M}}(f(p_\ell))$. So, it must be that

$$p_j \xrightarrow{to[x]} \in runs(\mathcal{T})$$

since $p_j \in \mathcal{E}^{\mathcal{T}}$. We thus have a contradiction as $j < \ell$ and

$$p_0 \xrightarrow{i_1 \cdots i_j \cdot to[x]} \in runs(\mathcal{T}). \qquad \square$$

Let us now prove Theorem 10.4.1, which we repeat one last time.

Theorem 10.4.1. *Let $\mathcal{M}$ be an s-learnable MMT and ζ be the length of the longest counterexample. Then,*

- *the $L^{\#}_{MMT}$ algorithm eventually terminates and returns an MMT $\mathcal{N}$ such that $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$ and whose size is polynomial in $|Q^{\mathcal{M}}|$ and factorial in $|X^{\mathcal{M}}|$, and*
- *in time and number of* $\mathbf{OQ}^{\mathrm{s}}, \mathbf{WQ}^{\mathrm{s}}, \mathbf{EQ}^{\mathrm{s}}$ *polynomial in $|Q^{\mathcal{M}}|$, $|I|$, and ζ, and factorial in $|X^{\mathcal{M}}|$.*

Proof. Let us start with showing that the algorithm eventually terminates. First, we formally prove that the refinement loop always finishes, *i. e.*, that the basis and the frontier always stabilize. By Lemma C.15.1 and Corollary C.15.2, we have

$$|\mathcal{B}^{\mathcal{T}}| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|} \tag{C.15.ii}$$

$$|\mathcal{F}^{\mathcal{T}}| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|} \cdot (|A(\mathcal{M})|) \tag{C.15.iii}$$

$$\max_{r \in \mathcal{F}^{\mathcal{T}}} |compat^{\mathcal{T}}(r)| \leq |Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|} \cdot |X^{\mathcal{M}}|! \tag{C.15.iv}$$

Furthermore, by the first part of Corollary 10.3.5

$$\max_{q \in Q^{\mathcal{T}}} |\chi^{\mathcal{T}}(q)| \leq |\chi^{\mathcal{T}}(f(q))| \leq |X^{\mathcal{M}}|. \tag{C.15.v}$$

Corollary 10.3.5. Let $\mathcal{T}$ be an observation tree for an s-learnable $\mathcal{M}$ with $\langle f, g\rangle$. Then, for all states $q \in Q^{\mathcal{T}}$, we have:

- $|\chi^{\mathcal{T}}(q)| \leq |\chi^{\mathcal{M}}(f(q))|$, and
- for all $x \in \chi_0^{\mathcal{T}}(q)$, $g(x) \in \chi_0^{\mathcal{M}}(f(q))$.

Let us argue that each part of the refinement loop is applied finitely many times. We write $|$**Seismic**$|$ for the number of times **Seismic** (see Section 10.4.4) is applied. We do Likewise for the other steps in the refinement loop.

- The maximal number of applied **Seismic** *per basis state* is bounded by $|X^{\mathcal{M}}|$, by (C.15.v). Indeed, each **Seismic** event is due to the discovery of a new active timer in a basis state. Hence, by (C.15.ii),

$$|\mathbf{Seismic}| \leq |\mathcal{B}^{\mathcal{T}}| \cdot |X^{\mathcal{M}}| \leq |Q^{\mathcal{M}}| \cdot |X^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|}. \tag{C.15.vi}$$

- By (C.15.ii), the number of times **Promotion** is applied between two instances of **Seismic** is bounded by $|Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|}$. So,

$$|\textbf{Promotion}| \leq |\mathcal{B}^{\mathcal{T}}| \cdot |\textbf{Seismic}| \leq |Q^{\mathcal{M}}|^2 \cdot |X^{\mathcal{M}}| \cdot 2^{2|X^{\mathcal{M}}|}. \tag{C.15.vii}$$

- Between two instances of **Seismic**, the number of **Completion** is bounded by $|I| \cdot |\mathcal{B}^{\mathcal{T}}|$, as, in the worst case, each input-transition is missing from each basis state. In general, we may have multiple frontier states $r_1, \dots, r_n$ such that $f(r_1) = \dots = f(r_n)$. Thus, $compat^{\mathcal{T}}(r_1) = \dots = compat^{\mathcal{T}}(r_n)$, after minimizing each set. Due to **Seismic**, Algorithm 10.1 potentially has to choose multiple times one of those states. So, in the worst case, we select a different r_j each time. As each new basis state may not have all of its outgoing transitions,

$$\begin{aligned} |\textbf{Completion}| &\leq |I| \cdot |\mathcal{B}^{\mathcal{T}}| \cdot |\textbf{Seismic}| \\ &= |I| \cdot |\textbf{Promotion}| \\ &\leq |I| \cdot |Q^{\mathcal{M}}|^2 \cdot |X^{\mathcal{M}}| \cdot 2^{2|X^{M}|}. \end{aligned} \tag{C.15.viii}$$

- Between two occurrences of **Seismic**, the number of pairs $(p, m) \in compat^{\mathcal{T}}(r)$ such that $|\chi^{\mathcal{T}}(p)| \neq |\chi^{\mathcal{T}}(r)|$ is directly given by (C.15.iv) for each frontier state r. So,

$$\begin{aligned} |\textbf{Active timers}| &\leq |\mathcal{F}^{\mathcal{T}}| \cdot \max_{r \in \mathcal{F}^{\mathcal{T}}} |compat^{\mathcal{T}}(r)| \cdot |\textbf{Seismic}| \\ &\leq |A(\mathcal{M})| \cdot |Q^{\mathcal{M}}|^3 \cdot |X^{\mathcal{M}}| \cdot 2^{3|X^{\mathcal{M}}|} \cdot |X^{\mathcal{M}}|!\,. \end{aligned} \tag{C.15.ix}$$

- With similar arguments,

$$\begin{aligned} |\textbf{WCT}| &\leq |\mathcal{F}^{\mathcal{T}}| \cdot \left(\max_{r \in \mathcal{F}^{\mathcal{T}}} |compat^{\mathcal{T}}(r)|\right)^2 \cdot |\textbf{Seismic}| \\ &\leq |A(\mathcal{M})| \cdot |Q^{\mathcal{M}}|^4 \cdot |X^{\mathcal{M}}|^2 \cdot 2^{4|X^{\mathcal{M}}|} \cdot (|X^{\mathcal{M}}|!)^2. \end{aligned} \tag{C.15.x}$$

Since each part of the refinement loop can only be applied a finite number of times, it follows that the loop always terminates. Thus, it remains to prove that Algorithm 10.1 constructs finitely many hypotheses. By Proposition 10.4.11, any counterexample results in a new timer being discovered for a state in the basis (leading to an occurrence of **Seismic**), or a compatibility set decreasing in size (potentially leading to a **Promotion**). We already know that |**Seismic**| and |**Promotion**| are bounded by a finite constant. Moreover, by (C.15.iv), each compatible set contains finitely many pairs. So, there can only be finitely many counterexamples, and, thus, hypotheses. More precisely, the number of hypotheses is bounded by

$$\begin{aligned} &|\textbf{Seismic}| \cdot |\textbf{Promotion}| \cdot \max_{r \in \mathcal{F}^{\mathcal{T}}} |compat^{\mathcal{T}}(r)| \\ &\leq |Q^{\mathcal{M}}|^4 \cdot |X^{\mathcal{M}}|^2 \cdot 2^{4|X^{\mathcal{M}}|} \cdot |X^{\mathcal{M}}|!\,. \end{aligned} \tag{C.15.xi}$$

To establish that $\mathcal{N}$ is equivalent to $\mathcal{M}$, we observe that the last equivalence query to the teacher confirmed they are symbolically equivalent. Hence, by Proposition 9.4.5, they are also timed equivalent, *i.e.*, $\mathcal{N} \overset{\text{time}}{\approx} \mathcal{M}$. From our construction of an MMT hypothesis based on an gMMT (see Section C.13), we

Proposition 10.4.11. When processing a counterexample, we eventually find a j such that $replay^{m_j^{-1}}_{\xrightarrow[r_j]{v_j'}}(p_j)$ returns APART or ACTIVE.

Proposition 9.4.5. Let $\mathcal{M}$ and $\mathcal{N}$ be two sound and complete MMTs. If $\mathcal{M} \overset{\text{sym}}{\approx} \mathcal{N}$, then $\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$.

get that the intermediate gMMT has at most $|Q^{\mathcal{M}}| \cdot 4^{|X^{\mathcal{M}}|}$ states (by (C.15.ii)). Hence, the final MMT has at most

$$|Q^{\mathcal{M}}| \cdot 2^{|X^{\mathcal{M}}|} \cdot |X^{\mathcal{M}}|!$$

states by Proposition C.13.5, *i. e.*, a number that is polynomial in $|Q^{\mathcal{M}}|$ and factorial in $|X^{\mathcal{M}}|$, as announced.

Proposition C.13.5. Let $\mathcal{M}$ be a sound and complete gMMT and $\mathcal{N}$ be the MMT constructed as explained above. Then, $\mathcal{N}$ is sound, complete, and its number of states is in $\mathcal{O}(n! \cdot |Q^{\mathcal{M}}|)$ with $n = \max_{q \in Q^{\mathcal{M}}} |\chi^{\mathcal{M}}(q)|$.

We now prove the claimed number of symbolic queries. First, we study the number of queries per step of the refinement loop, and to process a counterexample.

Seismic Applying **Seismic** does not require any symbolic queries.

Promotion Let r be the state newly added to $\mathcal{B}^{\mathcal{T}}$. We thus need to do a wait query in each r' such that $r \xrightarrow{i} r'$ for some $i \in A(\mathcal{T})$. By (C.15.v), there are at most $|A(\mathcal{M})|$ wait queries.

Completion A single application of **Completion** requires a single symbolic output query and a single wait query.

Active timers Each occurrence of **Active timers** necessitates to replay a run π from state q. Let n be the number of transitions in π. In the worst case, we have to perform n symbolic output queries and n symbolic wait queries (see Algorithm C.1). By Lemma C.15.3, $n \leq |Q^{\mathcal{M}}|$, if we always select a minimal run ending in the timeout of the desired timer.

WCT Likewise, applying **WCT** requires to replay a run of length n, *i. e.*, we do n symbolic output queries and n wait queries. This time, let us argue that we can always select a run such that

$$n \leq |\mathcal{B}^{\mathcal{T}}| + 1 + |Q^{\mathcal{M}}| + \zeta + (|\mathcal{B}^{\mathcal{T}}| + 1) \cdot |\textbf{Seismic}|.$$

(Recall that ζ is the length of the longest counterexample.) Let us decompose the summands appearing on the right piece by piece:

- $|\mathcal{B}^{\mathcal{T}}| + 1$ denotes the worst possible depth for a frontier state. Indeed, it may be that all basis states are on a single branch. So, the frontier states of the last basis state of that branch are at depth $|\mathcal{B}^{\mathcal{T}}| + 1$.
- $|Q^{\mathcal{M}}|$ comes from Lemma C.15.3, as (enabled) requires to see the timeout of some timer.
- ζ comes from the counterexample processing (see next item).
- When we previously replayed a witness of apartness due to some occurrences of **WCT**, we had to copy runs from a frontier state. Since the worst possible depth of a frontier state is $|\mathcal{B}^{\mathcal{T}}| + 1$, this means we added (at most) that length of the copied run when counted from the root of the observation tree. Since these replays may have triggered some instances of **Seismic**, the basis must have been recomputed each time. As explained above, we may not obtain the same exact basis, but the bound over the number of states is still (C.15.ii). So, in the worst case, we add $|\textbf{Seismic}|$ many times $(|\mathcal{B}^{\mathcal{T}}| + 1)$ to the longest branch of the tree.

Processing a counterexample First, in the worst case, we have to add the complete counterexample to observe what is needed (see Section 10.4.5), creating a new run in the tree, whose length is thus ζ.

Recall that each iteration of the counterexample processing splits a run $p \xrightarrow{v}$ with $p \in \mathcal{B}^{\mathcal{T}}$ into $p \xrightarrow{v'} r \xrightarrow{v''}$ such that $v = v' \cdot v''$ and $r \in \mathcal{F}^{\mathcal{T}}$. It may be that every v' is of length 1, meaning that we replay runs of lengths $\zeta, \zeta - 1, \dots, 1$. So, we do

$$\zeta + \zeta - 1 + \dots + 1 = \frac{\zeta^2 + \zeta}{2}$$

symbolic output queries and the same number of wait queries.

By combining with the bounds of (C.15.vi) to (C.15.x), we obtain the following bounds.

- The number of symbolic output queries is bounded by

$$1 \cdot |\mathbf{Completion}| + |Q^{\mathcal{M}}| \cdot |\mathbf{Active\ timers}| + \\ \left(|\mathcal{B}^{\mathcal{T}}| + 1 + |Q^{\mathcal{M}}| + \zeta + (|\mathcal{B}^{\mathcal{T}}| + 1) \cdot |\mathbf{Seismic}|\right) \cdot |\mathbf{WCT}|.$$

  Clearly, $(\zeta + |\mathcal{B}^{\mathcal{T}}| \cdot |\mathbf{Seismic}|) \cdot |\mathbf{WCT}|$ is bigger than the other operands (observe that ζ is independent from $|Q^{\mathcal{M}}|$, $|I|$, and $|X^{\mathcal{M}}|$). Hence, the number of symbolic output queries is in

$$\mathcal{O}\left(\left(\zeta + |Q^{\mathcal{M}}|^2 \cdot |X^{\mathcal{M}}| \cdot 2^{2|X^{\mathcal{M}}|}\right) \cdot |A(\mathcal{M})| \cdot |Q^{\mathcal{M}}|^4 \cdot \\ |X^{\mathcal{M}}|^2 \cdot 2^{4|X^{\mathcal{M}}|} \cdot (|X^{\mathcal{M}}|!)^2\right).$$

- The number of symbolic wait queries is bounded by

$$|A(\mathcal{M})| \cdot |\mathbf{Promotion}| + 1 \cdot |\mathbf{Completion}| + \\ |Q^{\mathcal{M}}| \cdot |\mathbf{Active\ timers}| + \\ \left(|\mathcal{B}^{\mathcal{T}}| + 1 + |Q^{\mathcal{M}}| + \zeta + (|\mathcal{B}^{\mathcal{T}}| + 1) \cdot |\mathbf{Seismic}|\right) \cdot |\mathbf{WCT}|.$$

  Again, $(\zeta + |\mathcal{B}^{\mathcal{T}}| \cdot |\mathbf{Seismic}|) \cdot |\mathbf{WCT}|$ is bigger than the other operands. That is, we obtain the same complexity results as for symbolic output queries.
- The number of symbolic equivalence queries is exactly the number of constructed hypothesis. It is thus bounded by (C.15.xi).

We thus obtain the announced complexity results. □

Part V.

Conclusion

Summary and future prospects 11.

To conclude this thesis, we give an overview of the contributions we presented throughout the document. We then describe some research directions related to the topic at hand, which we leave for future work.

Chapter contents

11.1. Summary

We studied active learning algorithms for automata. Our main contributions are two new learning algorithms for realtime one-counter automata and Mealy machines with timers, and a validation algorithm for JSON documents that first learns an automaton. In this section, we summarize these contributions.

First, in Part II, based on [BPS22], we studied and drew a hierarchy of multiple variants of one-counter automata. Importantly, we defined *realtime one-counter automata*, which are deterministic finite automata extended with a single natural counter and showed that its behavior can be represented in finite memory, despite the fact that the counter is unbounded (in general). To obtain this representation, we first define an infinite automaton, called the *behavior graph*, from a straightforward adaptation of the Myhill-Nerode equivalence relation. We showed that the behavior graph is isomorphic to the homonymous concept used in [NL10]. Given the results already proved by Neider and Löding, we immediately obtain the existence of a finite representation. The learner hence infers a sufficiently big finite fragment of the infinite automaton such that a finite representation of the target model can be extracted, and using a number of queries that is exponential in the number of input symbols, in the number of states of the teacher's ROCA, and in the length of the longest counterexample returned on an equivalence query. We implemented our algorithm and provided experimental results.

[BPS22]: Bruyère et al. (2022), "Learning Realtime One-Counter Automata"

[NL10]: Neider et al. (2010), *Learning visibly one-counter automata in polynomial time*

Second, in Part III, based on [BPS23], we provided a streaming validation algorithm that checks whether a JSON document satisfies some constraints given as a JSON schema. Our approach is to first construct a visibly pushdown automaton that accepts a (strict) subset of all valid documents. More precisely, the VPA assumes a fixed order on the key-value pairs appearing in the objects. As objects ought to be unordered, our algorithm removes this restriction by "jumping around" in the VPA to process each pair independently, using a key graph to know where to jump. Once the object has been fully read, we are

[BPS23]: Bruyère et al. (2023), "Validating Streaming JSON Documents with Learned VPAs" able to decide whether it satisfied the corresponding constraints. That is, we accept any permutation of key-value pairs, not only the learned order. We showed that our algorithm requires an amount of memory that is polynomial in the number of keys, in the number of states of the VPA, and in the depth (the number of nested objects and arrays) of the document, and has a time complexity that is exponential in the number of keys. Finally, our experimental results indicated that our algorithm usually takes more time than the classical algorithm (used in many applications) but far less memory, thanks to the fact that we do not have to hold the whole document in memory.

Finally, in Part IV, based on [Bru+23; Bru+24], we introduced Mealy machines with timers, which are a restriction of timed Mealy machines that use timers instead of clocks. We showed the reachability decision problem remains PSPACE-complete, and adapted the notions of regions and zones that are well-known tools for timed automata. Furthermore, we focused on the problem of deciding whether an MMT has an untimed run that can only be observed via some timed runs in which some delays must be zero, *i. e.*, whether races cannot be avoided. We showed that this decision problem is PSPACE-hard and provided an algorithm that is in 3EXP. Finally, we gave an active learning algorithm for MMTs, that is based on $L^{\#}$. This algorithm infers a finite tree and extracts an MMT from it, using a number of queries that is polynomial in the number of states of the teacher's MMT, in the number of input symbols, and in the length of the longest counterexample, and factorial in the number of timers of the teacher's MMT.

[Bru+23]: Bruyère et al. (2023), "Automata with Timers"
[Bru+24]: Bruyère et al. (2024), "Active Learning of Mealy Machines with Timers"

11.2. Future prospects

Let us conclude by giving several potential research directions that could extend the contributions presented throughout this document. Note that most of them were already given in their corresponding chapters.

11.2.1. Learning one-counter automata

Our algorithm for learning realtime one-counter automata, presented in Chapter 5, could be extended in the following ways:

- Remove the need of partial equivalence queries. In this direction, perhaps replacing our use of Neider and Löding's VOCA algorithm by Isberner's TTT algorithm [IHS14b; Isb15] might help, due to the differences in how the gathered knowledge is stored in both algorithms. The $L^{\#}$ algorithm of Vaandrager *et al.* [Vaa+22] is another potential candidate, as it does not infer an equivalence relation but instead focuses on identifying the differences in behavior of two states (see Section 3.3).
- Even without exploring the previous item, lowering the (query) complexity of our algorithm would help making it applicable to complex systems. One possible approach would be to change how a counterexample is processed. In [RS93], it is proved that adding a single separator after a failed equivalence query is enough to update the observation table and refine the knowledge of the learner. This would remove the suffix-closedness

[IHS14b]: Isberner et al. (2014), "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning"
[Isb15]: Isberner (2015), "Foundations of active automata learning: an algorithmic perspective"
[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"
[RS93]: Rivest et al. (1993), "Inference of Finite Automata Using Homing Sequences"

requirements on the separator sets S and $\hat{S}$. It is not immediately clear to us whether the definition of $\perp$-consistency presented here holds in that context. Further optimizations, such as discrimination tree-based algorithms (such as Kearns and Vazirani's algorithm [KV94]), also do not need the separator set to be suffix-closed.

[KV94]: Kearns et al. (1994), *An Introduction to Computational Learning Theory*

- Directly learn the one-counter language instead of an ROCA. Indeed, our algorithm learns some ROCA that accepts the target language. It would be desirable to learn some canonical representation of the language (*e.g.*, a minimal automaton, for some notion of minimality).
- Extend the learning algorithm to more general one-counter automata, such as *deterministic one-counter automata*, in which ε-transitions are allowed.[1]

1: See Section 4.2 for the definition.

Finally, while it is known that deciding whether two ROCAs are equivalent is NL-complete [BGJ14], there currently exists no concrete algorithms to do so. It may be an interesting (theoretical) question to tackle.

[BGJ14]: Böhm et al. (2014), "Bisimulation equivalence and regularity for real-time one-counter automata"

11.2.2. Validation of JSON documents

We now give some potential future work extending our validation algorithm for JSON documents, presented in Chapter 7:

- So far, our algorithm necessitates to learn a VPA, which takes a long time. Although this can be done once,[2] it may be interesting to directly construct a VPA from the JSON schema. While this task is easy if the schema does not contain Boolean operations, it is not yet clear how to proceed in the general case.

2: That is, once the VPA is computed, we never have to re-learn it.

- It could be worthwhile to compare our algorithm against an implementation of a classical algorithm used in the industry. This would require either to modify the industrial implementations to support abstractions, or to modify our algorithm to work on unabstracted JSON schemas.
- In our validation approach, we decided to use a VPA accepting the JSON documents satisfying a fixed key order — thus requiring to use the key graph and its costly computation of a set composed of the valid runs inside the VPA. It could be interesting to make additional experiments to compare this approach with one where we instead use a VPA accepting the JSON documents and all their key permutations — in this case, reasoning on the key graph would no longer be needed.
- Motivated by obtaining efficient querying algorithms on XML trees, the authors of [SSM08] have introduced the concept of mixed automata in a way to accept subsets of unranked trees where some nodes have ordered sons and some other have unordered sons. It would be interesting to adapt our validation algorithm to different formalisms of documents, such as the one of mixed automata.

[SSM08]: Seidl et al. (2008), "Counting in trees"

- Finally, exploring the possibility of using *systems of procedural automata* (SPAs, for short) [FS21], which form an extension of DFAs that can mutually call each other. The restrictions imposed by Frohme and Steffen for the learning algorithm (namely, that calls and returns between the different DFAs are observable) make the model akin to VPAs. As SPAs have specific structures (*i.e.*, distinct DFAs with special transitions indicating how to jump between them) and as the learning algorithm proposed

[FS21]: Frohme et al. (2021), "Compositional learning of mutually recursive procedural systems" in [FS21] has a lower time complexity than $\mathtt{TTT}_{\mathrm{VPL}}$, it may be interesting to adapt the ideas from our validation algorithm to SPAs, potentially yielding an algorithm with a better time complexity.

11.2.3. Mealy machines with timers

While timed automata are well-known (see Chapter 8) and the idea of using timers [Dil89] actually predates clocks, many properties of automata (or Mealy machines) with timers remain to be studied. We already showed in Chapter 9 that the reachability problem remains PSPACE-complete. The following questions may be worth exploring:

[Dil89]: Dill (1989), "Timing Assumptions and Verification of Finite-State Concurrent Systems"

- Is the set of automata with timers closed under classical operations (intersection, union, complementation, and so on)?
- We argued in Section 9.3 that there are timed automata that cannot be converted into an automaton with timers. The question is then: what is the exact subclass of timed automata for which it is possible?
- The MMT that we used in Section C.2 to show that the timed equivalence does not imply symbolic equivalence is *not* race-avoiding. Whether the implication holds when both MMTs are race-avoiding is an open question.
- We studied in Section 9.6 the problem of deciding whether it is possible to have concurrent actions in the timed runs of an automaton with timers. We proved that it is PSPACE-hard and provided a 3EXPalgorithm. What is the exact complexity class of the problem?

Our learning algorithm for Mealy machines with timers, presented in Chapter 10, could be extended in the following ways:

- The counterexample processing we described here is based on a linear search and reproduces parts of the runs until the tree is sufficiently modified to break the last constructed hypothesis. It may be interesting to perform a binary search (as is done in [Vaa+22], for instance) instead, which would reduce the number of required queries.
- In order to build a hypothesis MMT, we need to first construct a generalized MMT, in which timers can be arbitrarily renamed along the transitions. Hence, it would be worth to investigate whether it is possible to learn generalized Mealy machines with timers. This would require to adapt the arguments and tools introduced here.
  Another direction would be to provide a hypothesis construction that directly constructs an MMT, without needing generalized models.
- The current implementation always constructs a hypothesis MMT without going through generalized machines. This restricts the set of MMTs that can be learned. It would thus be interesting to implement the missing pieces.[3]
- Finally, one could apply the learning algorithm on concrete systems using more than two timers and perform model checking on the resulting MMTs (*e.g.*, by using UPPAAL [Beh+06]).

[Vaa+22]: Vaandrager et al. (2022), "A New Approach for Active Automata Learning Based on Apartness"

3: More details about the implementation will be available in the PhD thesis of Bharat Garhewal.

[Beh+06]: Behrmann et al. (2006), "UPPAAL 4.0"

11.2.4. Other directions

Finally, we list some other directions that are not (directly) related to the contributions presented in this thesis:

- Providing active learning algorithms for automata extended with other kinds of resources, such as *queue automata* [KMW18], or to extend existing works to more general models, *e.g.*, learning pushdown automata is currently restricted to the visibly case [IS14; MO22] and may benefit from more expressive subfamilies.
- To help with the previous item, studying *compositional learning* [FS21; NS23] where the learner infers the global system piece by piece before gluing them together may lead to better overall complexities for our algorithms.
- In parallel of timed automata, there exists a theory of *timed games*. See [MPS95; AHV93], for instance. It may be interesting to check whether considering timers instead of clocks could help obtaining better complexity results in this domain.
- The contributions presented in this thesis were mostly theoretical. Applying the proposed algorithms on concrete examples coming from the industry and formally verifying the resulting models is an interesting direction. In particular, this may help us designing tools that can easily be used to model check real-world systems.

[KMW18]: Kutrib et al. (2018), “Queue Automata: Foundations and Developments”

[IS14]: Isberner et al. (2014), “An Abstract Framework for Counterexample Analysis in Active Automata Learning”

[MO22]: Michaliszyn et al. (2022), “Learning Deterministic Visibly Pushdown Automata Under Accessible Stack”

[FS21]: Frohme et al. (2021), “Compositional learning of mutually recursive procedural systems”

[NS23]: Neele et al. (2023), “Compositional Automata Learning of Synchronous Systems”

[MPS95]: Maler et al. (1995), “On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract)”

[AHV93]: Alur et al. (1993), “Parametric real-time reasoning”

Bibliography

[Aar+15] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. "Generating models of infinite-state communication protocols using regular inference with abstraction". In: *Formal Methods in System Design* 46.1 (2015), pp. 1–41. DOI: 10.1007/S10703-014-0216-X (cit. on p. 39).

[AD94] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8 (cit. on pp. 183, 187, 188, 191, 192, 213, 233).

[AF16] Dana Angluin and Dana Fisman. "Learning regular omega languages". In: *Theoretical Computer Science* 650 (2016), pp. 57–72. DOI: 10.1016/J.TCS.2016.07.031 (cit. on p. 6).

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. "Parametric real-time reasoning". In: *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. Ed. by S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal. ACM, 1993, pp. 592–601. DOI: 10.1145/167088.167242 (cit. on p. 339).

[Aic+18] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. "Model Learning and Model-Based Testing". In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*. Ed. by Amel Bennaceur, Reiner Hähnle, and Karl Meinke. Vol. 11026. Lecture Notes in Computer Science. Springer, 2018, pp. 74–100. DOI: 10.1007/978-3-319-96562-8_3 (cit. on pp. 3, 6).

[AL02] Luca Aceto and François Laroussinie. "Is your model checker on time? On the complexity of model checking for timed modal logics". In: *Journal of Logic and Algebraic Programming* 52-53 (2002), pp. 7–51. DOI: 10.1016/S1567-8326(02)00022-X (cit. on pp. 184, 187, 274).

[Alu+05] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. "Congruences for Visibly Pushdown Languages". In: *Proceedings of the 32nd International Colloquium Automata, Languages and Programming, ICALP 2005*. Ed. by Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung. Vol. 3580. Lecture Notes in Computer Science. Springer, 2005, pp. 1102–1114. DOI: 10.1007/11523468_89 (cit. on pp. 136, 139–141).

[Alu99] Rajeev Alur. "Timed Automata". In: *Proceedings of the 11th International Conference Computer Aided Verification, CAV 1999*. Ed. by Nicolas Halbwachs and Doron A. Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 8–22. DOI: 10.1007/3-540-48683-6_3 (cit. on p. 213).

[AM04] Rajeev Alur and P. Madhusudan. "Visibly pushdown languages". In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*. Ed. by László Babai. ACM, 2004, pp. 202–211. DOI: 10.1145/1007352.1007390 (cit. on pp. 45, 137, 140, 148).

[An+20] Jie An, Mingshuai Chen, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. "Learning One-Clock Timed Automata". In: *Proceedings of the 26th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2020*. Ed. by Armin Biere and David Parker. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 444–462. DOI: 10.1007/978-3-030-45190-5_25 (cit. on pp. 5, 184, 193, 233, 267).

[An+21] Jie An, Lingtai Wang, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. “Learning real-time automata”. In: *Science China Information Sciences* 64.9 (2021). DOI: 10.1007/S11432-019-2767-4 (cit. on p. 184).

[Ang87] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Information and Computation* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6 (cit. on pp. 3, 4, 18–20, 24, 25, 39, 59, 64, 78, 101, 136, 141, 143, 193, 235).

[APT20] Bernhard K. Aichernig, Andrea Pferscher, and Martin Tappler. “From Passive to Active: Learning Timed Automata Efficiently”. In: *Proceedings of the 12th International Symposium NASA Formal Methods, NFM 2020*. Ed. by Ritchie Lee, Susmit Jha, and Anastasia Mavridou. Vol. 12229. Lecture Notes in Computer Science. Springer, 2020, pp. 1–19. DOI: 10.1007/978-3-030-55754-6_1 (cit. on pp. 184, 193, 233, 265, 267).

[AR16] Andreas Abel and Jan Reineke. “Gray-Box Learning of Serial Compositions of Mealy Machines”. In: *Proceedings of the 8th International Symposium NASA Formal Methods, NFM 2016*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 272–287. DOI: 10.1007/978-3-319-40648-0_21 (cit. on pp. 5, 40).

[Baa+19] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. “Schemas And Types For JSON Data”. In: *Proceedings of the 22nd International Conference on Extending Database Technology Advances in Database Technology, EDBT 2019*. Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi. OpenProceedings.org, 2019, pp. 437–439. DOI: 10.5441/002/EDBT.2019.39 (cit. on p. 93).

[BBM21] Henrik Björklund, Johanna Björklund, and Wim Martens. “Learning algorithms”. In: *Handbook of Automata Theory*. Ed. by Jean-Éric Pin. European Mathematical Society Publishing House, Zürich, Switzerland, 2021, pp. 375–409. DOI: 10.4171/AUTOMATA-1/11 (cit. on p. 3).

[BCS15] Iovka Boneva, Radu Ciucanu, and Slawek Staworko. “Schemas for Unordered XML on a DIME”. In: *Theoretical Computer Science* 57.2 (2015), pp. 337–376. DOI: 10.1007/S00224-014-9593-1 (cit. on p. 126).

[Beh+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. “UPPAAL 4.0”. In: *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, QEST 2006*. IEEE Computer Society, 2006, pp. 125–126. DOI: 10.1109/QEST.2006.59 (cit. on pp. 183, 338).

[Ber+21] Raphaël Berthon, Adrien Boiret, Guillermo A. Pérez, and Jean-François Raskin. “Active Learning of Sequential Transducers with Side Information About the Domain”. In: *Proceedings of the 25th International Conference Developments in Language Theory, DLT 2021*. Ed. by Nelma Moreira and Rogério Reis. Vol. 12811. Lecture Notes in Computer Science. Springer, 2021, pp. 54–65. DOI: 10.1007/978-3-030-81508-0_5 (cit. on pp. 5, 40).

[BF72] Alan W. Biermann and Jerome A. Feldman. “On the Synthesis of Finite-State Machines from Samples of Their Behavior”. In: *IEEE Transactions on Computers* 21.6 (1972), pp. 592–597. DOI: 10.1109/TC.1972.5009015 (cit. on pp. 3, 18).

[BGJ14] Stanislav Böhm, Stefan Göller, and Petr Jancar. “Bisimulation equivalence and regularity for real-time one-counter automata”. In: *Journal of Computer and System Sciences* 80.4 (2014), pp. 720–743. DOI: 10.1016/J.JCSS.2013.11.003 (cit. on pp. 59, 95, 337).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9 (cit. on pp. 1, 2, 6, 183, 184, 213).

[Bol16] Benedikt Bollig. “One-Counter Automata with Counter Observability”. In: *Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016*. Ed. by Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen. Vol. 65. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 20:1–20:14. DOI: 10.4230/LIPICS.FSTTCS.2016.20 (cit. on p. 58).

[Bou+11] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. “Programs with lists are counter automata”. In: *Formal Methods in System Design* 38.2 (2011), pp. 158–192. DOI: 10.1007/S10703-011-0111-7 (cit. on p. 39).

[Bou+18] Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, Joël Ouaknine, and James Worrell. “Model Checking Real-Time Systems”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 1001–1046. DOI: 10.1007/978-3-319-10575-8_29 (cit. on pp. 183, 193).

[Bou+22] Patricia Bouyer, Paul Gastin, Frédéric Herbreteau, Ocan Sankur, and B. Srivathsan. “Zone-Based Verification of Timed Automata: Extrapolations, Simulations and What Next?” In: *Proceedings of the 20th International Conference Formal Modeling and Analysis of Timed Systems, FORMATS 2022*. Ed. by Sergiy Bogomolov and David Parker. Vol. 13465. Lecture Notes in Computer Science. Springer, 2022, pp. 16–42. DOI: 10.1007/978-3-031-15839-1_2 (cit. on p. 189).

[BPS22] Véronique Bruyère, Guillermo A. Pérez, and Gaëtan Staquet. “Learning Realtime One-Counter Automata”. In: *Proceedings of the 28th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 244–262. DOI: 10.1007/978-3-030-99524-9_13 (cit. on pp. 6, 8, 58, 102, 335).

[BPS23] Véronique Bruyère, Guillermo A. Pérez, and Gaëtan Staquet. “Validating Streaming JSON Documents with Learned VPAs”. In: *Proceedings of the 29th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13993. Lecture Notes in Computer Science. Springer, 2023, pp. 271–289. DOI: 10.1007/978-3-031-30823-9_14 (cit. on pp. 7, 8, 126, 135, 164, 335).

[BR87] Piotr Berman and Robert Roos. “Learning One-Counter Languages in Polynomial Time (Extended Abstract)”. In: *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1987, pp. 61–67. DOI: 10.1109/SFCS.1987.36 (cit. on p. 59).

[Bra17] Tim Bray. “The JavaScript Object Notation (JSON) Data Interchange Format”. In: *RFC* 8259 (2017), pp. 1–16. DOI: 10.17487/RFC8259 (cit. on pp. 97, 127, 128).

[Bru+23] Véronique Bruyère, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager. “Automata with Timers”. In: *Proceedings of the 21st International Conference Formal Modeling and Analysis of Timed Systems, FORMATS 2023*. Ed. by Laure Petrucci and Jeremy Sproston. Vol. 14138. Lecture Notes in Computer Science. Springer, 2023, pp. 33–49. DOI: 10.1007/978-3-031-42626-1_3 (cit. on pp. 7, 8, 192, 268, 336).

[Bru+24] Véronique Bruyère, Bharat Garhewal, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager. “Active Learning of Mealy Machines with Timers”. In: *CoRR* abs/2403.02019 (2024). DOI: 10.48550/ARXIV.2403.02019 (cit. on pp. 7, 8, 192, 232, 268, 336).

[BY03] Johan Bengtsson and Wang Yi. “Timed Automata: Semantics, Algorithms and Tools”. In: *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer, 2003, pp. 87–124. DOI: `10.1007/978-3-540-27755-2\_3` (cit. on pp. 183, 189).

[Cer92] Karlis Cerans. “Decidability of Bisimulation Equivalences for Parallel Timer Processes”. In: *Proceedings of the 4th International Workshop Computer Aided Verification, CAV 1992*. Ed. by Gregor von Bochmann and David K. Probst. Vol. 663. Lecture Notes in Computer Science. Springer, 1992, pp. 302–315. DOI: `10.1007/3-540-56496-9\_24` (cit. on p. 188).

[Cho78] Tsun S. Chow. “Testing Software Design Modeled by Finite-State Machines”. In: *IEEE transactions on software engineering* 4.3 (1978), pp. 178–187. DOI: `10.1109/TSE.1978.231496` (cit. on p. 4).

[Cla+18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1 (cit. on pp. 2, 6, 183, 184, 188, 189, 213, 225).

[CR04] Cristiana Chitic and Daniela Rosu. “On Validation of XML Streams Using Finite State Machines”. In: *Proceedings of the 7th International Workshop on the Web and Databases, WebDB 2004*. Ed. by Sihem Amer-Yahia and Luis Gravano. ACM, 2004, pp. 85–90. DOI: `10.1145/1017074.1017096` (cit. on pp. 39, 59, 97).

[Daw+95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. “The Tool KRONOS”. In: *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems*. Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Vol. 1066. Lecture Notes in Computer Science. Springer, 1995, pp. 208–219. DOI: `10.1007/BFB0020947` (cit. on p. 265).

[DD17] Samuel Drews and Loris D’Antoni. “Learning Symbolic Automata”. In: *Proceedings of the 23rd International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 173–189. DOI: `10.1007/978-3-662-54577-5\_10` (cit. on p. 6).

[Die+23] Simon Dierl, Falk Maria Howar, Sean Kauffman, Martin Kristjansen, Kim Guldstrand Larsen, Florian Lorber, and Malte Mauritz. “Learning Symbolic Timed Models from Concrete Timed Data”. In: *Proceedings of the 15th International Symposium NASA Formal Methods, NFM 2023*. Ed. by Kristin Yvonne Rozier and Swarat Chaudhuri. Vol. 13903. Lecture Notes in Computer Science. Springer, 2023, pp. 104–121. DOI: `10.1007/978-3-031-33170-1\_7` (cit. on p. 193).

[Dil89] David L. Dill. “Timing Assumptions and Verification of Finite-State Concurrent Systems”. In: *Proceedings of the International Workshop Automatic Verification Methods for Finite State Systems*. Ed. by Joseph Sifakis. Vol. 407. Lecture Notes in Computer Science. Springer, 1989, pp. 197–212. DOI: `10.1007/3-540-52148-8\_17` (cit. on pp. 193, 338).

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *Computer* 11.4 (1978), pp. 34–41. DOI: `10.1109/C-M.1978.218136` (cit. on p. 154).

[Fer+21] Tiago Ferreira, Harrison Brewton, Loris D’Antoni, and Alexandra Silva. “Prognosis: closed-box analysis of network protocol implementations”. In: *Proceedings of the ACM SIGCOMM 2021 Conference*. Ed. by Fernando A. Kuipers and Matthew C. Caesar. ACM, 2021, pp. 762–774. DOI: `10.1145/3452296.3472938` (cit. on p. 3).

[FFZ23] Dana Fisman, Hadar Frenkel, and Sandra Zilles. “Inferring Symbolic Automata”. In: *Lecture Notes in Computer Science* 19.2 (2023). DOI: 10.46298/LMCS-19(2:5)2023 (cit. on p. 6).

[FH17] Paul Fiterau-Brostean and Falk Howar. “Learning-Based Testing the Sliding Window Behavior of TCP Implementations”. In: *Proceedings of the Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems FMICS and 17th International Workshop on Automated Verification of Critical Systems AVoCS 2017*. Ed. by Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti. Vol. 10471. Lecture Notes in Computer Science. Springer, 2017, pp. 185–200. DOI: 10.1007/978-3-319-67113-0_12 (cit. on p. 3).

[Fit+17] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. “Model learning and model checking of SSH implementations”. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. Ed. by Hakan Erdogmus and Klaus Havelund. ACM, 2017, pp. 142–151. DOI: 10.1145/3092282.3092289 (cit. on p. 3).

[Fit+20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. “Analysis of DTLS Implementations Using Protocol State Fuzzing”. In: *Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 2523–2540. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean (cit. on p. 3).

[FJV16] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. “Combining Model Learning and Model Checking to Analyze TCP Implementations”. In: *Proceedings of the 28th International Conference Computer Aided Verification, CAV 2016*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 454–471. DOI: 10.1007/978-3-319-41540-6_25 (cit. on p. 3).

[FMR68] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. “Counter Machines and Counter Languages”. In: *Mathematical Systems Theory* 2.3 (1968), pp. 265–283. DOI: 10.1007/BF01694011 (cit. on pp. 40, 63).

[FR95] Amr F. Fahmy and Robert S. Roos. “Efficient Learning of Real Time One-Counter Automata”. In: *Proceedings of the 6th International Conference Algorithmic Learning Theory, ALT 1995*. Ed. by Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann. Vol. 997. Lecture Notes in Computer Science. Springer, 1995, pp. 25–40. DOI: 10.1007/3-540-60454-5_26 (cit. on pp. 5, 44, 59, 60).

[FS18] Markus Frohme and Bernhard Steffen. “Active Mining of Document Type Definitions”. In: *Proceedings of the 23rd International Conference Formal Methods for Industrial Critical Systems, FMICS 2018*. Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 147–161. DOI: 10.1007/978-3-030-00244-2_10 (cit. on p. 163).

[FS21] Markus Frohme and Bernhard Steffen. “Compositional learning of mutually recursive procedural systems”. In: *International Journal on Software Tools for Technology Transfer* 23.4 (2021), pp. 521–543. DOI: 10.1007/S10009-021-00634-Y (cit. on pp. 163, 337–339).

[Gar+20] Bharat Garhewal, Frits W. Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. “Grey-Box Learning of Register Automata”. In: *Proceedings of the 16th International Conference Integrated Formal Methods, IFM 2020*. Ed. by Brijesh Dongol and Elena Troubitsyna. Vol. 12546. Lecture Notes in Computer Science. Springer, 2020, pp. 22–40. DOI: 10.1007/978-3-030-63461-2_2 (cit. on pp. 5, 40).

[GJL04] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. "Learning of Event-Recording Automata". In: *Proceedings of the Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems - Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004.* Ed. by Yassine Lakhnech and Sergio Yovine. Vol. 3253. Lecture Notes in Computer Science. Springer, 2004, pp. 379–396. DOI: 10.1007/978-3-540-30206-3_26 (cit. on p. 232).

[GJL10] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. "Learning of event-recording automata". In: *Theoretical Computer Science* 411.47 (2010), pp. 4029–4054. DOI: 10.1016/J.TCS.2010.07.008 (cit. on pp. 184, 193, 233).

[GJP06] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. "Inference of Event-Recording Automata Using Timed Decision Trees". In: *Proceedings of the 17th International Conference Concurrency Theory, CONCUR 2006.* Ed. by Christel Baier and Holger Hermanns. Vol. 4137. Lecture Notes in Computer Science. Springer, 2006, pp. 435–449. DOI: 10.1007/11817949_29 (cit. on pp. 193, 233).

[GPY06] Alex Groce, Doron A. Peled, and Mihalis Yannakakis. "Adaptive Model Checking". In: *Logic Journal of the IGPL* 14.5 (2006), pp. 729–744. DOI: 10.1093/JIGPAL/JZL007 (cit. on pp. 6, 39).

[GTW03] Erich Grädel, Wolfgang Thomas, and Thomas Wilke. *Automata, logics, and infinite games: a guide to current research.* Vol. 2500. Springer, 2003 (cit. on pp. 221, 287).

[Hen+92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. "Symbolic Model Checking for Real-time Systems". In: *Proceedings of the 7th Annual Symposium on Logic in Computer Science, LICS 1992.* IEEE Computer Society, 1992, pp. 394–406. DOI: 10.1109/LICS.1992.185551 (cit. on p. 189).

[HJM20] Léo Henry, Thierry Jéron, and Nicolas Markey. "Active Learning of Timed Automata with Unobservable Resets". In: *Proceedings of the 18th International Conference Formal Modeling and Analysis of Timed Systems, FORMATS 2020.* Ed. by Nathalie Bertrand and Nils Jansen. Vol. 12288. Lecture Notes in Computer Science. Springer, 2020, pp. 144–160. ISBN: 978-3-030-57627-1 (cit. on pp. 184, 233).

[HS18] Falk Howar and Bernhard Steffen. "Active Automata Learning in Practice - An Annotated Bibliography of the Years 2011 to 2016". In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers.* Ed. by Amel Bennaceur, Reiner Hähnle, and Karl Meinke. Vol. 11026. Lecture Notes in Computer Science. Springer, 2018, pp. 123–148. DOI: 10.1007/978-3-319-96562-8_5 (cit. on pp. 3, 193).

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. ISBN: 0-201-02988-X (cit. on pp. 14, 15, 48, 49, 51, 273).

[HV10] Marijn Heule and Sicco Verwer. "Exact DFA Identification Using SAT Solvers". In: *Proceedings of the 10th International Conference on Grammatical Inference, ICGI 2010.* Ed. by José M. Sempere and Pedro García. Vol. 6339. Lecture Notes in Computer Science. Springer, 2010, pp. 66–79. DOI: 10.1007/978-3-642-15488-1_7 (cit. on pp. 3, 18).

[IHS14a] Malte Isberner, Falk Howar, and Bernhard Steffen. "Learning register automata: from languages to program structures". In: *Machine Learning* 96.1-2 (2014), pp. 65–98. DOI: 10.1007/S10994-013-5419-7 (cit. on p. 5).

[IHS14b] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". In: *Proceedings of the 5th Runtime Verification, RV 2014*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Vol. 8734. Lecture Notes in Computer Science. Springer, 2014, pp. 307–322. DOI: 10.1007/978-3-319-11164-3_26 (cit. on pp. 20, 100, 141, 336).

[IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib - A Framework for Active Automata Learning". In: *Proceedings of the 27th International Conference Computer Aided Verification, CAV 2015*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 487–495. DOI: 10.1007/978-3-319-21690-4_32 (cit. on p. 94).

[IS14] Malte Isberner and Bernhard Steffen. "An Abstract Framework for Counterexample Analysis in Active Automata Learning". In: *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014*. Ed. by Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka. Vol. 34. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 79–93. URL: http://proceedings.mlr.press/v34/isberner14a.html (cit. on pp. 25, 339).

[Isb15] Malte Isberner. "Foundations of active automata learning: an algorithmic perspective". PhD thesis. Technical University Dortmund, Germany, 2015. URL: https://hdl.handle.net/2003/34282 (cit. on pp. 5, 7, 8, 100, 135, 136, 141, 142, 155, 336).

[JH11] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE transactions on software engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62 (cit. on p. 154).

[Joh87] Marjory J. Johnson. "Proof that Timing Requirements of the FDDI Token Ring Protocol are Satisfied". In: *IEEE Transactions on Computers* 35.6 (1987), pp. 620–625. DOI: 10.1109/TCOM.1987.1096832 (cit. on p. 265).

[JV18] Bengt Jonsson and Frits Vaandrager. *Learning mealy machines with timers*. Tech. rep. Radboud University, Nijmegen, 2018 (cit. on p. 233).

[Khm+22] Igor Khmelnitsky, Serge Haddad, Lina Ye, Benoît Barbot, Benedikt Bollig, Martin Leucker, Daniel Neider, and Rajarshi Roy. "Analyzing Robustness of Angluin's L* Algorithm in Presence of Noise". In: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022*. Ed. by Pierre Ganty and Dario Della Monica. Vol. 370. EPTCS. 2022, pp. 81–96. DOI: 10.4204/EPTCS.370.6 (cit. on p. 5).

[KKG23] Paul Kogel, Verena Klös, and Sabine Glesner. "Learning Mealy Machines with Local Timers". In: *Proceedings of the 24th International Conference on Formal Engineering Methods Formal Methods and Software Engineering, ICFEM 2023*. Ed. by Yi Li and Sofiène Tahar. Vol. 14308. Lecture Notes in Computer Science. Springer, 2023, pp. 47–64. DOI: 10.1007/978-981-99-7584-6_4 (cit. on pp. 194, 265, 267).

[KMV07] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. "Visibly pushdown automata for streaming XML". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*. Ed. by Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy. ACM, 2007, pp. 1053–1062. DOI: 10.1145/1242572.1242714 (cit. on pp. 126, 136, 165, 168).

[KMW18] Martin Kutrib, Andreas Malcher, and Matthias Wendlandt. "Queue Automata: Foundations and Developments". In: *Reversibility and Universality, Essays Presented to Kenichi Morita on the Occasion of his 70th Birthday*. Ed. by Andrew Adamatzky. Vol. 30. Emergence, Complexity and Computation. Springer, 2018, pp. 385–431. DOI: 10.1007/978-3-319-73216-9_19 (cit. on p. 339).

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994. ISBN: 978-0-262-11193-5. URL: https://mitpress.mit.edu/books/introduction-computational-learning-theory (cit. on pp. 20, 101, 337).

[LN12] Martin Leucker and Daniel Neider. “Learning Minimal Deterministic Automata from Inexperienced Teachers”. In: *Proceedings of the 5th International Symposium Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, ISoLA 2012.* Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 7609. Lecture Notes in Computer Science. Springer, 2012, pp. 524–538. DOI: 10.1007/978-3-642-34026-0_39 (cit. on pp. 4, 59, 76, 77).

[LY96] David Lee and Mihalis Yannakakis. “Principles and methods of testing finite state machines-a survey”. In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123. DOI: 10.1109/5.533956 (cit. on p. 19).

[Mar+17] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. “BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema”. In: *ACM Transactions on Database Systems* 42.3 (2017), 15:1–15:42. DOI: 10.1145/3105960 (cit. on p. 126).

[MO20] Jakub Michaliszyn and Jan Otop. “Learning Deterministic Automata on Infinite Words”. In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI 2020.* Ed. by Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 2370–2377. DOI: 10.3233/FAIA200367 (cit. on pp. 5, 40).

[MO22] Jakub Michaliszyn and Jan Otop. “Learning Deterministic Visibly Pushdown Automata Under Accessible Stack”. In: *Proceedings of the 47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022.* Ed. by Stefan Szeider, Robert Ganian, and Alexandra Silva. Vol. 241. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 74:1–74:16. DOI: 10.4230/LIPICS.MFCS.2022.74 (cit. on p. 339).

[MP04] Oded Maler and Amir Pnueli. “On Recognizable Timed Languages”. In: *Proceedings of the 7th International Conference Foundations of Software Science and Computation Structures, FOSSACS 2004.* Ed. by Igor Walukiewicz. Vol. 2987. Lecture Notes in Computer Science. Springer, 2004, pp. 348–362. DOI: 10.1007/978-3-540-24727-2_25 (cit. on pp. 232–234).

[MP95] Oded Maler and Amir Pnueli. “On the Learnability of Infinitary Regular Sets”. In: *Information and Computation* 118.2 (1995), pp. 316–326. DOI: 10.1006/INCO.1995.1070 (cit. on p. 6).

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. “On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract)”. In: *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1995.* Ed. by Ernst W. Mayr and Claude Puech. Vol. 900. Lecture Notes in Computer Science. Springer, 1995, pp. 229–242. DOI: 10.1007/3-540-59042-0_76 (cit. on p. 339).

[Nei14] Daniel Neider. “Applications of automata learning in verification and synthesis”. PhD thesis. RWTH Aachen University, 2014. URL: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2014/5169 (cit. on pp. 3, 18).

[NJ13] Daniel Neider and Nils Jansen. “Regular Model Checking Using Solver Technologies and Automata Learning”. In: *Proceedings of the 5th Internation Symposium NASA Formal Methods, NFM 2013.* Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 16–31. DOI: 10.1007/978-3-642-38088-4_2 (cit. on pp. 3, 18).

[NL10] Daniel Neider and Christof Löding. *Learning visibly one-counter automata in polynomial time*. Tech. rep. Technical Report AIB-2010-02, RWTH Aachen (January 2010), 2010 (cit. on pp. 5, 6, 8, 39, 40, 44–46, 51–59, 74, 91, 114, 118, 124, 335).

[NS18] Matthias Niewerth and Thomas Schwentick. "Reasoning About XML Constraints Based on XML-to-Relational Mappings". In: *Theoretical Computer Science* 62.8 (2018), pp. 1826–1879. DOI: `10.1007/S00224-018-9846-5` (cit. on p. 126).

[NS23] Thomas Neele and Matteo Sammartino. "Compositional Automata Learning of Synchronous Systems". In: *Proceedings of the 26th International Conference Fundamental Approaches to Software Engineering, FASE 2023*. Ed. by Leen Lambers and Sebastián Uchitel. Vol. 13991. Lecture Notes in Computer Science. Springer, 2023, pp. 47–66. DOI: `10.1007/978-3-031-30826-0\_3` (cit. on p. 339).

[OG92] osé Oncina and Pedro Garcia. "Inferring regular languages in polynomial updated time". In: *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific. 1992, pp. 49–61. DOI: `10.1142/9789812797902_0004` (cit. on p. 3).

[Pet11] Holger Petersen. "Simulations by Time-Bounded Counter Machines". In: *International Journal of Foundations of Computer Science* 22.2 (2011), pp. 395–409. DOI: `10.1142/S0129054111008106` (cit. on p. 40).

[Pez+16] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. "Foundations of JSON Schema". In: *Proceedings of the 25th International Conference on World Wide Web, WWW 2016*. Ed. by Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao. ACM, 2016, pp. 263–273. DOI: `10.1145/2872427.2883029` (cit. on pp. 127, 129, 132, 133, 135).

[PVY02] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. "Black Box Checking". In: *Journal of Automata, Languages and Combinatorics* 7.2 (2002), pp. 225–246. DOI: `10.25596/JALC-2002-225` (cit. on pp. 6, 39).

[Roo88] Robert Stewart Roos. *Deciding equivalence of deterministic one-counter automata in polynomial time with applications to learning*. The Pennsylvania State University, 1988 (cit. on p. 59).

[RP15] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *Proceedings of the 24th USENIX Security Symposium, USENIX Security 15*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, 2015, pp. 193–206. URL: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter` (cit. on p. 3).

[RS93] Ronald L. Rivest and Robert E. Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Information and Computation* 103.2 (1993), pp. 299–347. DOI: `10.1006/INCO.1993.1021` (cit. on pp. 25, 101, 336).

[San23] Ocan Sankur. "Timed Automata Verification and Synthesis via Finite Automata Learning". In: *Proceedings of 29th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 329–349. DOI: `10.1007/978-3-031-30820-8\_21` (cit. on pp. 6, 184).

[Sch12] Thomas Schwentick. "Foundations of XML Based on Logic and Automata: A Snapshot". In: *Proceedings of the 7th International Symposium Foundations of Information and Knowledge Systems, FoIKS 2012*. Ed. by Thomas Lukasiewicz and Attila Sali. Vol. 7153. Lecture Notes in Computer Science. Springer, 2012, pp. 23–33. DOI: `10.1007/978-3-642-28472-4\_2` (cit. on p. 126).

[SG09] Muzammil Shahbaz and Roland Groz. “Inferring Mealy Machines”. In: *Proceedings of the 16th Formal Methods, FM 2009*. Ed. by Ana Cavalcanti and Dennis Dams. Vol. 5850. Lecture Notes in Computer Science. Springer, 2009, pp. 207–222. DOI: 10.1007/978-3-642-05089-3_14 (cit. on pp. 25, 26, 236).

[SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. “Introduction to Active Automata Learning from a Practical Perspective”. In: *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011*. Ed. by Marco Bernardo and Valérie Issarny. Vol. 6659. Lecture Notes in Computer Science. Springer, 2011, pp. 256–296. DOI: 10.1007/978-3-642-21455-4_8 (cit. on p. 26).

[SSM08] Helmut Seidl, Thomas Schwentick, and Anca Muscholl. “Counting in trees”. In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*. Ed. by Jörg Flum, Erich Grädel, and Thomas Wilke. Vol. 2. Texts in Logic and Games. Amsterdam University Press, 2008, pp. 575–612 (cit. on pp. 163, 337).

[SV02] Luc Segoufin and Victor Vianu. “Validating Streaming XML Documents”. In: *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. Ed. by Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis. ACM, 2002, pp. 53–64. DOI: 10.1145/543613.543622 (cit. on pp. 126, 136).

[Tan+22] Xiaochen Tang, Wei Shen, Miaomiao Zhang, Jie An, Bohua Zhan, and Naijun Zhan. “Learning Deterministic One-Clock Timed Automata via Mutation Testing”. In: *Proceedings of the 20th International Symposium Automated Technology for Verification and Analysis, ATVA 2022*. Ed. by Ahmed Bouajjani, Lukás Holík, and Zhilin Wu. Vol. 13505. Lecture Notes in Computer Science. Springer, 2022, pp. 233–248. DOI: 10.1007/978-3-031-19992-9_15 (cit. on p. 5).

[Tan09] Nguyen Van Tang. “A Tighter Bound for the Determinization of Visibly Pushdown Automata”. In: *Proceedings of the International Workshop on Verification of Infinite-State Systems, INFINITY 2009*. Ed. by Axel Legay. Vol. 10. EPTCS. 2009, pp. 62–76. DOI: 10.4204/EPTCS.10.5 (cit. on p. 140).

[Tap+19] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. “Time to Learn - Learning Timed Automata from Tests”. In: *Proceedings of the 17th International Conference Formal Modeling and Analysis of Timed Systems, FORMATS 2019*. Ed. by Étienne André and Mariëlle Stoelinga. Vol. 11750. Lecture Notes in Computer Science. Springer, 2019, pp. 216–235. DOI: 10.1007/978-3-030-29662-9_13 (cit. on p. 233).

[Tho97] Wolfgang Thomas. “Languages, Automata, and Logic”. In: *Handbook of Formal Languages, Volume 3: Beyond Words*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Springer, 1997, pp. 389–455. DOI: 10.1007/978-3-642-59126-6_7 (cit. on pp. 221, 287).

[TY01] Stavros Tripakis and Sergio Yovine. “Analysis of Timed Systems Using Time-Abstracting Bisimulations”. In: *Formal Methods in System Design* 18.1 (2001), pp. 25–68. DOI: 10.1023/A:1008734703554 (cit. on pp. 188, 213).

[TZA24] Yu Teng, Miaomiao Zhang, and Jie An. “Learning Deterministic Multi-Clock Timed Automata”. In: *CoRR* abs/2404.07823 (2024). DOI: 10.48550/ARXIV.2404.07823 (cit. on pp. 5, 184).

[Vaa+22] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. “A New Approach for Active Automata Learning Based on Apartness”. In: *Proceedings of the 28th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 223–243. DOI: 10.1007/978-3-030-99524-9_12 (cit. on pp. 19, 26–28, 30, 33, 35, 100, 233, 236, 238, 239, 242, 247, 336, 338).

[Vaa17] Frits W. Vaandrager. “Model learning”. In: *Communications of the ACM* 60.2 (2017), pp. 86–95. DOI: 10.1145/2967606 (cit. on p. 3).

[VBE21] Frits W. Vaandrager, Roderick Bloem, and Masoud Ebrahimi. “Learning Mealy Machines with One Timer”. In: *Proceedings of the 15th International Conference Language and Automata Theory and Applications, LATA 2021*. Ed. by Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron. Vol. 12638. Lecture Notes in Computer Science. Springer, 2021, pp. 157–170. DOI: 10.1007/978-3-030-68195-1_13 (cit. on pp. 193, 194, 265, 267).

[VP75] Leslie G. Valiant and Mike Paterson. “Deterministic One-Counter Automata”. In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 340–350. DOI: 10.1016/S0022-0000(75)80005-5 (cit. on p. 60).

[VWW07] Sicco Verwer, Mathijs De Weerdt, and Cees Witteveen. “An algorithm for learning real-time automata”. In: *Proceeding of the Annual Belgian-Dutch Machine Learning Conference, Benelearn 2007*. 2007 (cit. on p. 184).

[Wag23] Masaki Waga. “Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization”. In: *Proceedings of the 35th International Conference Computer Aided Verification, CAV 2023*. Ed. by Constantin Enea and Akash Lal. Vol. 13964. Lecture Notes in Computer Science. Springer, 2023, pp. 3–26. DOI: 10.1007/978-3-031-37706-8_1 (cit. on pp. 5, 184, 193, 233, 265–267).

[WDR13] Md Tawhid Bin Waez, Jürgen Dingel, and Karen Rudie. “A survey of timed automata for the development of real-time systems”. In: *Computer Science Review* 9 (2013), pp. 1–26. DOI: 10.1016/J.COSREV.2013.05.001 (cit. on pp. 187, 191).

[XAZ22] Runqing Xu, Jie An, and Bohua Zhan. “Active Learning of One-Clock Timed Automata Using Constraint Solving”. In: *Proceedings of the 20th International Symposium Automated Technology for Verification and Analysis, ATVA 2022*. Ed. by Ahmed Bouajjani, Lukás Holík, and Zhilin Wu. Vol. 13505. Lecture Notes in Computer Science. Springer, 2022, pp. 249–265. DOI: 10.1007/978-3-031-19992-9_16 (cit. on pp. 233, 267).

Index

Notations

Notation	**Description**
$w \vdash q \# p$	The states q and p are apart, with w a witness.
$f \circ g$	The function such that $(f \circ g)(x) = f(g(x))$.
ε	The unique word with zero symbol.
$\equiv_{\mathcal{O}}$	The equivalence relation of the observation table $\mathcal{O}$.
$\mathcal{M} \approx \mathcal{N}$	Mealy machines $\mathcal{M}$ and $\mathcal{N}$ are equivalent: for every input word, both $\mathcal{M}$ and $\mathcal{N}$ produce the same output word.
$[\![\cdot]\!]_{\sim}$	An equivalence class of the relation $\sim$.
$\cong$	The region relation.
$J \models \mathsf{S}$	The JSON value J satisfies the JSON schema given as a non-terminal symbol S in the extended context-free grammar.
$\sim_L$	The Myhill-Nerode congruence for the language L.
$\sim_{\mathcal{A}}$	The refined Myhill-Nerode congruence for a realtime one-counter automaton $\mathcal{A}$.
$\simeq_L$	The Myhill-Nerode congruence for visibly pushdown languages.
$\widetilde{\Sigma}$	A pushdown alphabet.
$\widetilde{\Sigma}_{\text{JSON}}$	The pushdown alphabet used in visibly pushdown automata for JSON schemas..
$\mathcal{M} \overset{\text{time}}{\approx} \mathcal{N}$	Mealy machines with timers $\mathcal{M}$ and $\mathcal{N}$ are timed equivalent.
$A(\mathcal{M})$	The set of actions of a Mealy machine with timers $\mathcal{M}$ comprised of all considered input and timeout symbols.
$\mathit{Approx}(v)$	The approximation set of v in L^*_{ROCA}.
$\beta(u)$	The balance of u, *i. e.*, the number of unmatched call (when the value is greater than zero) or return (when the value is smaller than zero) symbols in u.
$\mathcal{B}^{\mathcal{T}}$	The basis of the observation tree $\mathcal{T}$ in $L^{\#}$.
$BG(L)$	The behavior graph of the visibly one-counter language L.
$BG(\mathcal{A})$	The behavior graph of the realtime one-counter automaton $\mathcal{A}$.
$BG_{\leq \ell}(\mathcal{A})$	The bounded behavior graph up to ℓ of the realtime one-counter automaton $\mathcal{A}$.
$\mathit{compat}^{\mathcal{T}}(r)$	The set of all basis states that are compatible with the frontier state r.
$\mathsf{cnstr}(\pi)$	The constraints accumulated along the (untimed) run π of a Mealy machine with timers.
$\mathsf{CP}(\widetilde{\Sigma})$	The context pairs over the pushdown alphabet $\widetilde{\Sigma}$.
$\mathit{cruns}(\mathcal{A})$	The set of counted runs of the one-counter automaton $\mathcal{A}$.
$\mathit{cv}(w)$	The counter value of a word w over a pushdown alphabet.
$\mathit{cv}^{\mathcal{A}}(w)$	The counter value of a word w for a realtime one-counter automaton $\mathcal{A}$.
$\mathbf{CVQ}(w)$	A counter value query: gets $\mathit{cv}^{\mathcal{A}}(w)$, with $\mathcal{A}$ the teacher's realtime one-counter automaton.
$\mathsf{dom}(f)$	The domain of a function $f : A \rightharpoonup B$, *i. e.*, the set of values $a \in A$ such that $f(a)$ is defined.
$\mathbf{EQ}(\mathcal{H})$	An equivalence query: returns **yes** when $\mathcal{H}$ accepts the target language, or a counterexample otherwise.

Notation	**Description**
$\mathbf{EQ}^{\mathbf{s}}(\mathcal{H})$	A symbolic equivalence query for $L^{\#}_{\mathrm{MMT}}$: returns **yes** when $\mathcal{H}$ accepts the target language, or a counterexample otherwise.
$\mathcal{F}^{\mathcal{T}}$	The frontier of the observation tree $\mathcal{T}$ in $L^{\#}$.
$\mathcal{G}$	The extended context-free grammar for a JSON schema.
$\mathcal{G}_{\mathrm{U}}$	The universal extended context-free grammar.
$\mathrm{G}_{\mathcal{A}}$	The key graph of the 1-SEVPA $\mathcal{A}$.
$\mathit{height}(w)$	The height of a word w over a pushdown alphabet.
$\mathit{height}^{\mathcal{A}}(w)$	The height of a word w for a realtime one-counter automaton $\mathcal{A}$.
$\mathbb{I}_A$	The identity relation over the set A, *i.e.*, $\mathbb{I}_A = \{(a,a) \mid a \in A\}$.
$\mathcal{L}(\mathcal{A})$	The language of the automaton $\mathcal{A}$.
$L_{\leq \ell}$	The language of the bounded behavior graph up to ℓ of a visibly or realtime one-counter automaton.
$\mathcal{L}_{\leq \ell}(\mathcal{A})$	The bounded language up to ℓ of the realtime one-counter automaton $\mathcal{A}$.
$\mathcal{L}_{<}(\mathcal{G})$	The subset of $\mathcal{L}(\mathcal{G})$ where keys inside objects respect the order $<$..
$m^{\pi}_{\pi'}$	The runs π and π' of a Mealy machine with timers match, according to the matching m.
$\mathbf{MQ}(w)$	A membership query: check whether w belongs to the target language.
$\mathcal{O}$	An observation table for L^*.
$\mathcal{O}_{\leq \ell}$	An observation table up to ℓ for L^*_{VOCA} and L^*_{ROCA}.
$\mathit{output}^{\mathcal{M}}(w)$	The output word obtained by concatenating the output symbols of the run of w in $\mathcal{M}$.
$\mathbf{OQ}(w)$	An output query: gets $\mathit{output}^{\mathcal{M}}(w)$, with $\mathcal{M}$ the Mealy machine of the teacher..
$\mathbf{OQ}^{\mathbf{s}}(w)$	A symbolic output query for $L^{\#}_{\mathrm{MMT}}$: gets $\mathit{toutputs}^{\mathcal{M}}(w)$, with w a symbolic input word and $\mathcal{M}$ the Mealy machine with timers of the teacher..
$\mathit{ptruns}(\mathcal{M})$	The set of padded timed runs of the Mealy machine with timers $\mathcal{M}$.
$\mathbf{PEQ}(\mathcal{H}, \ell)$	A partial equivalence query: returns **yes** when $\mathcal{H}$ accepts the bounded target language up to ℓ, or a counterexample otherwise.
$\mathit{Pref}(\cdot)$	The set of prefixes of a word or language.
$\mathsf{ran}(f)$	The range of a function $f : A \rightharpoonup B$, *i.e.*, the set of values $b \in B$ such that $f(a) = b$ for some $a \in \mathsf{dom}(f)$.
$\mathrm{Reach}_{\mathcal{A}}$	The reachability relation of a visibly pushdown automaton.
$\overline{R}$	The set of representatives without $\bot$-words in L^*_{ROCA}.
$\mathit{runs}(\mathcal{A})$	The set of runs of the automaton or Mealy machine $\mathcal{A}$.
$\mathit{sign}(a)$	The sign (*i.e.*, the counter operation) of a symbol in a pushdown alphabet.
$\mathit{Suff}(\cdot)$	The set of suffixes of a word or language.
$\mathit{truns}(\mathcal{M})$	The set of timed runs of the Mealy machine with timers $\mathcal{M}$.
$\mathit{TO}[X]$	The set of timeout symbols $\mathit{to}[x]$ for each timer x of X.

Notation	**Description**
$tiw(\rho)$	The timed input word obtained from the delays and input symbols of the timed run ρ.
tiwruns(w)	The set of all timed runs induced by the timed input word w.
toutputs(w)	The set of timed input words obtained from every timed run induced by the timed input word w.
$tow(\rho)$	The timed output word obtained from the delays and output symbols of the timed run ρ.
$\mathcal{T}$	An observation tree for $L^{\#}$.
untime(π)	The untimed projection of the timed run π.
$U(\mathcal{M})$	The set of updates of $\mathcal{M}$ comprised of $\perp$ (no update), and of all pairs (x, c) with x a timer and c a natural.
$\mathbf{WQ^s}(w)$	A symbolic wait query for $L^{\#}_{\text{MMT}}$: gets the set of enabled timers after a run reading the symbolic input word w in the teacher's Mealy machine with timers..
zone$(\mathcal{M})$	The zone Mealy machine with timers of the Mealy machine with timers $\mathcal{M}$.

Abbreviations

Notation	**Description**
1-SEVPA	1-module single entry visibly pushdown automata.
DFA	deterministic finite automaton.
DOCA	deterministic one-counter automaton.
DOCL	deterministic one-counter language.
gMMT	generalized Mealy machine with timers.
LBTM	Linear-bounded Turing machine.
MM	Mealy machine.
MMT	Mealy machine with timers.
NFA	nondeterministic finite automaton.
OCA	one-counter automaton.
OCL	one-counter language.
ROCA	realtime one-counter automaton.
ROCL	realtime one-counter language.
sw	symbolic word.
tiw	timed input word.
TMM	timed Mealy machine.
tow	timed output word.
VOCA	visibly one-counter automaton.
VOCL	visibly one-counter language.
VPA	visibly pushdown automaton.
VPL	visibly pushdown language.