UMONS
Université de Mons

Faculté Polytechnique

Mutually Recursive Procedural Systems

Streaming Validation of JSON Documents

Master's Thesis
Submitted in partial fulfillment of the requirements for the master's degree in engineering in computer science and management with a focus on Cybersecurity and Forensics for the Internet of Everything.

Ir. Kévin DUBRULLE

POLYTECH MONS

Under the supervision of :
Ms. Véronique BRUYÈRE
Mr. Gaëtan STAQUET

June 2025

pôle hainuyer

Acknowledgements

I would like to thank all the people who supported me throughout the development of this Master's thesis.

First and foremost, I would like to thank my supervisors, Ms. Véronique Bruyère and Mr. Gaëtan Staquet. It was through countless discussions and email exchanges that this work took shape.

I would also like to thank Anne-Sophie Frans and Florian Dubois, who both helped me proofread this thesis to ensure that it was clear and written in proper English.

I am also grateful to Mr. Philippe Fortemps and Ms. Véronique Van Renterghem, without whom I would not have been able to pursue this second Master's degree, and consequently, this Master's thesis.

Finally, I would like to thank my family and friends for their emotional support without them, these past years would have felt much longer.

We refer to *Factorisation Non-Négative de Matrices Creuses en Norme ℓ_1 - Extraction de Thèmes et Classification Non-Supervisée de Textes*, June 2023, by Kévin Dubrulle, for more complete acknowledgements.

Abstract

Simple automated systems can often be modeled by finite automata. However, traditional automata have limited computational power, making them unsuitable for modeling more complex systems. In this work, we propose an extension of automata where multiple automata can mutually call each other recursively, with a mechanism to control non-deterministic procedural calls.

Our work focuses on JSON documents, which are widely used as a format for data exchange and storage, and easily readable and writable by both humans and machines. To ensure consistency and proper communication, JSON documents must conform to a certain structure, defined by a JSON Schema. Machines need to verify efficiently if the document satisfies the schema, with limited computation time and memory usage. The classical validation algorithm processes the entire document recursively according to the schema's constraints, requiring the full document to be loaded in memory. An alternative is based on a streaming algorithm, which reads the symbols one by one and consumes less memory. However, for complex schemas, its structure lacks the flexibility to handle it, and it becomes hard to understand the model.

We introduce a novel validation model – based on the extension of automata – that retains the streaming approach while mirroring the recursive structure of JSON schemas. This approach improves both efficiency of validation and readability of the model. Our model could be adapted to the validation of other types of recursively structured data, even in environments with limited computational resources.

Keywords :
Automaton – Procedural System – JSON document – Streaming Validation

Contents

List of Figures

List of Definitions

List of Algorithms

List of Abbreviations

DFA	Deterministic Finite Automaton
NFA	Non-deterministic Finite Automaton
PDA	Pushdown Automaton
VPA	Visibly Pushdown Automaton
SPA	System of Procedural Automaton
VSPA	Visibly System of Procedural Automaton
CFG	Context Free Grammar
CFL	Context Free Language
JSON	JavaScript Object Notation
API	Application Programming Interface
DFS	Depth First Search
HTML	HyperText Markup Language
UML	Unified Modeling Language
XML	Extensible Markup Language

Chapter 1

Introduction and Preliminaries

1.1 Introduction

Simple automated systems can usually be modeled by an automaton, which is a finite state machine describing the execution of the system. For instance, UML state machines are a widely-used example of extension of automata [15]. However, traditional automata have limited computational power because of the lack of memory beyond their current state. Allowing an automaton to use some resources – counters [11], stacks [12] or clocks [1] – is useful to model more complex systems.

Consider, for example, *JavaScript Object Notation* (JSON) documents. JSON is a standard format for data exchange, particularly in web applications, since it is designed to be easy for developers to read and write, and efficiently processed by machines. To ensure proper communication between two machines, the JSON document must conform to a certain structure, called *JSON schemas*.

When a machine receives a JSON document, it needs to check whether it satisfies the constraints imposed by the JSON schema. Usually, these kinds of machines have limited resources, or must handle large volumes of documents. Therefore, it is important to *validate* the documents efficiently, with low computation time and memory usage.

The classical approach to validate a JSON document involve recursively checking whether the documents respect the schema's rules. This can consume significant memory, especially for large documents. Moreover, a poorly designed schema can increase the computation time required to validate the document [6].

There already exists an algorithm that performs *streaming validation* of JSON documents, based on automaton [8]. It uses a limited amount of memory compared to a classical validation algorithm. The automaton used is *learned*, meaning that it is not directly derived from the JSON schema. This has certain advantages, as it is not affected by poor schema design. However, the automata used are not specifically optimized for the recursive nature of JSON schemas.

In this work, we propose a new automaton model – called *Visibly System of Procedural Automata* – that can perform streaming validation of JSON documents. This model extends an existing framework of automata that mutually call each other [10], incorporating some control for non-determinism of procedural calls. The proposed model should be well-suited for documents with a recursive structure, such as JSON documents.

In this chapter, we present some theoretical background necessary to understand the model, and we introduce the general notations used. This includes the theory of languages, automata, and the structure of JSON documents and schemas.

The second chapter focuses on the formal presentation of the model used to recursively validate documents. We outline some properties of the model and prove its applicability for the case of JSON schemas.

Finally, the third chapter details the practical implementation of the streaming validation algorithm. Additionally, we provide experimental results and compare our algorithm with the classical JSON validator and with the other existing streaming algorithm.

1.2 Language and Finite Automaton

This section explains the basis of language and automaton theory. This includes most of the notions and notations that will be used throughout this work. Most of the content of this section comes from the book *Introduction to Automata Theory, Language and Computation* by Hopcroft and Ullman [12].

1.2.1 Alphabet, Word and Language

A *symbol* is an abstract entity with no formal definition. Letters and digits are frequently used symbols. We can also use a symbol to represent an object, an operation... A finite non-empty set of symbols is called an *alphabet*. As an example, $\Sigma = \{a, b\}$ is an alphabet composed of two symbols : "a" and "b".

A *word* is a finite sequence of symbols. For example, $w = aaba$ is a word over the alphabet $\Sigma = \{a, b\}$. The length of a word w, noted $|w|$, is the number of symbols in the word. In the previous example, $|w| = |aaba| = 4$. The empty word is noted ε. It is composed of zero symbol, such that $|\varepsilon| = 0$.

The *concatenation* of two words is the word composed of the first one followed by the second one. The operation of the concatenation is noted $\cdot$ or is omitted. As an example, let $w_1 = ab$ and $w_2 = aa$ be two words over $\Sigma = \{a, b\}$. The concatenation of w_1 and w_2 is $w_1 \cdot w_2 = w_1 w_2 = abaa$.

We use exponent notation to denote the repetition of a symbol or a word. Here are some examples of words over the alphabet $\Sigma = \{a, b\}$ that use this notation : $a^4 = aaaa$, $ab^3a = abbba$, $(ab)^2 = abab$, $(bbaa)^0 = \varepsilon$.

A set of words over an alphabet is called a *language*. The empty set $\emptyset$ and the set composed of only the empty string $\{\varepsilon\}$ are two distinct languages. A specific language is the set of all words over an alphabet Σ (including the empty word ε). This language is noted Σ^*.

Here are some examples of languages over the alphabet $\Sigma = \{a, b\}$:

- $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, ...\}$.
- $L_{palindrome}$, the language of all palindromes over Σ. Some words in $L_{palindrome}$ are : ε, aa, bab.
- For a given word $w \in \Sigma^*$, $Pref(w) = \{u \in \Sigma^* | \exists v \in \Sigma^*, w = u \cdot v\}$ is the language of all prefixes of the word w (including the empty word ε). Similarly, the language of all suffixes of w is formally defined by $Suff(w) = \{v \in \Sigma^* | \exists u \in \Sigma^*, w = u \cdot v\}$.
- $L = \{w \in \Sigma^* | a \in Suff(w)\}$, the language of all words ending by the symbol a. Some words in this language are : a, ba, a^3. Note that the empty word ε is not in this language.
- $L = \{a^n b^n | n \in \mathbb{N}\}$, some words in this language are : $aabb$, ε, $a^5 b^5$.

1.2.2 Finite Automaton and Regular Language

Deterministic Finite Automaton – DFA

In computer science, an automaton is a state system that takes a word $w \in \Sigma^*$ as input and returns true if the word belongs to a specific language $L \subseteq \Sigma^*$. An automaton starts in a specific state and reads the symbols of w one by one. When it reads a symbol, the automaton may switch its current state, according to a transition function. After reading the whole word, if the automaton is in a *final* state, we say that the automaton accepts the word w (meaning that $w \in L$).

Definition 1.1: Deterministic Finite Automaton

A *Deterministic Finite Automaton* (DFA) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where :

- Σ is the input alphabet ;
- Q is the finite non-empty set of states, with $q_0 \in Q$ the initial state;
- $F \subseteq Q$ is the set of final states;
- $\delta : (Q \times \Sigma) \to Q$ is the complete transition function. We write $\delta(q, a) = p$ or $q \xrightarrow{a} p$ to denote a transition in δ, with $p, q \in Q$ and $a \in \Sigma$. The transition function δ is said complete if it is defined for every pair $(q, a) \in (Q \times \Sigma)$.

The transition function δ only describes the behavior of a DFA when it reads a symbol. We can extend it with the run function $\hat{\delta}$, which will describe the behavior of a DFA when it reads a word. The run of a word corresponds to the succession of transitions of each symbol of this word.

Definition 1.2: Run Function

The *run function* of a DFA is the function $\hat{\delta} : (Q \times \Sigma^*) \to Q : \hat{\delta}(q,w) = p$. It defines the state p reached starting from a state q after reading the whole word w. It is defined recursively from the transition function δ of the DFA :

1. $\hat{\delta}(q,\varepsilon) = q, \forall q \in Q$;
2. $\hat{\delta}(q,w \cdot a) = \delta\left(\hat{\delta}(q,w), a\right)$, with $q \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$.

A run $\hat{\delta}(q,w) = p$ can be written as $q \xrightarrow{w} p$.
For a word $w = a_1 ... a_n$, the run $q \xrightarrow{w=a_1...a_n} p$ can be decomposed as the succession of transitions for each symbol of w : $q \xrightarrow{a_1} ... \xrightarrow{a_n} p$.

With the definition of a run, it is easy to formally define when a word is accepted by the DFA. A word is accepted if, when the DFA reads the whole word, starting from the initial state q_0, it ends in a final state. This means that the run of an accepted word starting from q_0 is in the set of final states F.

Definition 1.3: Language of a DFA

We say that a word w is *accepted* by a DFA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ if the run of the word starting in the initial state q_0 ends in a final state : $\hat{\delta}(q_0,w) \in F$.
The *Language of a DFA* $\mathcal{A}$, noted $L(\mathcal{A})$, is the set of words accepted by $\mathcal{A}$:

$$L(\mathcal{A}) = \left\{ w \in \Sigma^* | \hat{\delta}(q_0,w) \in F \right\}.$$

A DFA $\mathcal{A}$ can be represented as an oriented graph, where the states in Q are the vertices of the graph and the transitions in δ are the edges of the graph.

An example of DFA $\mathcal{A}$ over the alphabet $\Sigma = \{a,b\}$ is shown in Figure 1.1a. The DFA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ is described by :

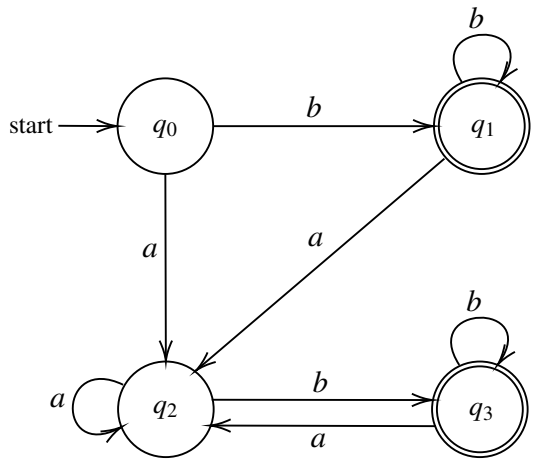
- $\Sigma = \{a,b\}$ is the input alphabet ;
- $Q = \{q_0, q_1, q_2, q_3\}$ is the set of four states, with q_0 the initial state ;
- $F = \{q_1, q_3\}$ is the set of final states (represented with double-circle edges in the graph) ;
- The transition function $\delta : (Q \times \Sigma) \to Q$ is described by the table in Figure 1.1b.

Here are some examples of runs in the DFA $\mathcal{A}$:

- $w_1 = baba$:

$$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{a} q_2 \Rightarrow q_0 \xrightarrow{w_1} q_2,$$

  the word w_1 is not accepted by $\mathcal{A}$ because the run $\hat{\delta}(w_1, q_0) = q_2 \notin F$.



(a) Example of DFA $\mathcal{A}$ over $\Sigma = \{a, b\}$.

δ		**Symbol** a	b
State	q_0	q_2	q_1
	q_1	q_2	q_1
	q_2	q_2	q_3
	q_3	q_2	q_3

(b) Transition Function δ of the DFA $\mathcal{A}$.

Figure 1.1: Example of DFA and its Transition Function.

- $w_3 = aabb$:

$$q_0 \xrightarrow{a} q_2 \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{b} q_3 \Rightarrow q_0 \xrightarrow{w_3} q_3,$$

  the word w_3 is accepted by $\mathcal{A}$ because the run $\hat{\delta}(w_3, q_0) = q_3 \in F$.

One can show that the language of the DFA $\mathcal{A}$ is the following :

$$L(\mathcal{A}) = \{w \in \Sigma^* | b \in Suff(w))\},$$

meaning that a word is accepted by $\mathcal{A}$ if it ends by b. Note that we can construct a 2-state DFA accepting the same language.

Regular Expression and Regular Language

A *regular expression* is a way to describe some languages. To define the construction of all regular expressions, we first need to define the concatenation operation of languages. Let L_1 and L_2 be two languages. The concatenation of L_1 and L_2, noted L_1L_2 is the set $\{u \cdot v | u \in L_1, v \in L_2\}$. Moreover, for any $i \in \mathbb{N}$, we note $L^i = LL^{i-1}$, with $L^0 = \{\varepsilon\}$ and $L^1 = L$.

We define the language L^* as the set of all words that can be formed with the concatenation of words from L, including the empty word ε. This language can be formally defined by : $L^* = \bigcup_{i \in \mathbb{N}} L^i$. Note that we already used this notation with Σ^*, the set of all words over an alphabet Σ.

Definition 1.4: Regular Expression and Regular Language

Let Σ be an alphabet. The *Regular Expressions* and the sets they denote, called *Regular Languages*, are recursively defined as follows :

- $\emptyset$ is a regular expression and denotes the empty set, which is a regular language.
- ε is a regular expression and denotes the regular language $\{\varepsilon\}$.
- For all $a \in \Sigma$, a is a regular expression and denotes the regular language $\{a\}$.
- Let l_1 and l_2 be two regular expressions, denoting respectively the regular languages L_1 and L_2. Then $l_1 + l_2$ is a regular expression and denotes the regular language $L_1 \cup L_2$.
- Let l_1 and l_2 be two regular expressions, denoting respectively the regular languages L_1 and L_2. Then $l_1 l_2$ is a regular expression and denotes the regular language $L_1 L_2 = \{v \cdot u | v \in L_1, w \in L_2\}$.
- Let l be a regular expression denoting the regular language L. Then l^* is a regular expression and denotes the regular language $L^* = \bigcup_{i \in \mathbb{N}} L^i$.

Here are some examples of regular languages over the alphabet $\Sigma = \{a, b, c\}$ and the regular expression denoting these languages :

- $L = \Sigma^*$ is denoted by the regular expression $(a+b+c)^*$;
- $L = \{u \cdot a \cdot v | u, v \in \Sigma^*\}$ is denoted by the regular expression $(a+b+c)^* a (a+b+c)^*$;
- $L = \{a^n b^m | n, m \in \mathbb{N}\}$ is denoted by the regular expression $a^* b^*$;
- $L = \{c^n \cdot a \cdot c^m \cdot b \cdot c^5 | n, m \in \mathbb{N}, n \geq 3, m \geq 1\}$ is denoted by the regular expression $c^3 c^* a c c^* b c^5$.

We can show that the set of all regular languages and the set of all languages accepted by a DFA are the same sets. This means that for every regular language, we can build a DFA accepting this language, and conversely.

Theorem 1.1 ([12]). *L is a regular language $\iff$ it exists a DFA $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*

It is important to note that there exist some languages that are not regular. Thus, we cannot build a DFA accepting these languages. Here are some examples of nonregular languages :

- $L = \{a^n b^n | n \in \mathbb{N}\}$. Note that this language is not to be confused with the language $\{a^n b^m | n, m \in \mathbb{N}\}$, which is regular ;
- L is a language over the alphabet $\Sigma = \{(,)\}$ such that every word $w \in L$ is well-parenthesized. L is a nonregular language. For example, $w = (()()) \in L$ and $w' = ())()(\notin L$.
- $L_{palindrome}$, the language of all palindromes over an alphabet Σ, is a nonregular language.

Non-deterministic Finite Automaton – NFA

The definition of DFAs requires that the transition function δ is complete. This means that for any word w, there always exists a run $\hat{\delta}(q_0, w)$, and this run is always unique. In a *Non-deterministic Finite Automaton*, it can exist zero, one or more transitions from a state for the same symbol. This means that for a word w, it can exist zero, one or more runs $\hat{\delta}(q_0, w)$.

Definition 1.5: Non-deterministic Finite Automaton

A *Non-deterministic Finite Automaton* (NFA) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where :

- Σ is the input alphabet ;
- Q is the finite non-empty set of states, with $q_0 \in Q$ the initial state;
- $F \subseteq Q$ is the set of final states;
- $\delta : (Q \times \Sigma) \to 2^Q$ is the transition function, where 2^Q represents the set of all possible subsets of Q, including the emptyset. A transition $q \xrightarrow{a} p$ is in the NFA $\mathcal{A}$ when $p \in \delta(q, a)$.

Similarly to the transition function, the run function $\hat{\delta}(q, w)$ gives the set of all states reachable when reading a word w from a state q : $\hat{\delta}(q, w) \in 2^Q$. A state p is reachable from a state q by reading a word $w = a_1 ... a_n$ if it exists a path $q \xrightarrow{a_1} ... \xrightarrow{a_n} p$ in the NFA : $p \in \hat{\delta}(q, w)$.

Definition 1.6: Language of NFA

A word w is accepted by an NFA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ if it exists a path starting from the initial state q_0 to some final state $q \in F$.
The *Language of an NFA* $\mathcal{A}$, noted $L(\mathcal{A})$, is the set of words accepted by $\mathcal{A}$:

$$w \in L(\mathcal{A}) \iff \exists q \in F \text{ s.t. } q \in \hat{\delta}(q_0, w),$$

$$w \in L(\mathcal{A}) \iff \exists q_0 \xrightarrow{w} q, \text{ with } q \in F.$$

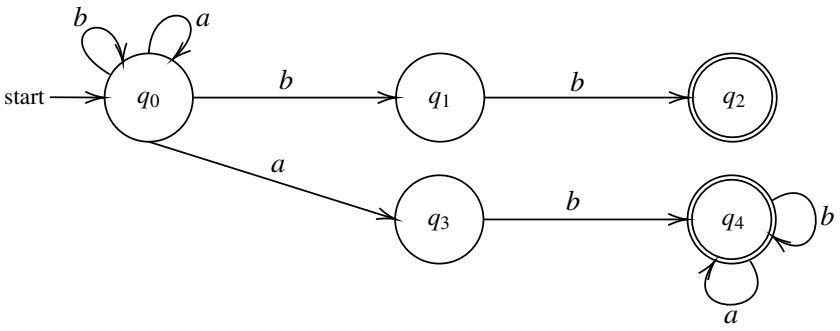
Like we did for DFAs, an NFA can be represented as an oriented graph, where the vertices are the states of the NFA and the edges are the transitions. Unlike DFA, we will see that from a same state, there can be more than one transition using the same symbol.

Figure 1.2a shows an example of a 5-state-NFA $\mathcal{A}$ over the alphabet $\Sigma = \{a, b\}$. Its transition function δ is described as a table in Figure 1.2b.

Here are two words and some possible paths of these words for the NFA $\mathcal{A}$:

- $w_1 = aba$, here are two possible paths for this word :
  - $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1$, and no transition reads the symbol a from q_1 ;
  - $q_0 \xrightarrow{a} q_3 \xrightarrow{b} q_4 \xrightarrow{a} q_4$.

  As $q_4 \in \hat{\delta}(q_0, w_1)$ and $q_4 \in F$, the word w_1 is accepted by $\mathcal{A}$.



(a) Example of NFA $\mathcal{A}$ over $\Sigma = \{a, b\}$.

δ		**Symbol**	
		a	b
State	q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
	q_1	$\emptyset$	$\{q_2\}$
	q_2	$\emptyset$	$\emptyset$
	q_3	$\emptyset$	$\{q_4\}$
	q_4	$\{q_4\}$	$\{q_4\}$

(b) Transition Function δ of the NFA $\mathcal{A}$.

Figure 1.2: Example of NFA and its Transition Function.

- $w_2 = bba$, here are all the possible paths for this word:
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{b} q_2$, and no transition reads the symbol a from q_2 ;
  - $q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_1$, and no transition reads the symbol a from q_1 ;
  - $q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_3$;
  - $q_0 \xrightarrow{b} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0$.

  As there is no final state among $\hat{\delta}(q_0, w_2) = \{q_3, q_0\}$, w is not accepted.

We can show that the language accepted by the NFA $\mathcal{A}$ is the following language :

$$L(\mathcal{A}) = \{w \in \Sigma^* | (bb \in Suff(w)) \vee (\exists u, v \in \Sigma^*, u \cdot ab \cdot v = w)\}.$$

This means that a word is accepted by $\mathcal{A}$ if it ends by bb or if it contains the sequence ab. The following regular expression describes the language $L(\mathcal{A})$: $((a+b)^*bb) + ((a+b)^*ab(a+b)^*)$.

We can ask ourselves what class of languages is accepted by NFAs compared to DFAs. It is clear that any DFA can be represented as an NFA, as a DFA is just an NFA where, for all pairs (q, a), the transition function $\delta(q, a)$ is a set containing exactly one state. However, it turns out that any language accepted by an NFA can also be accepted by a DFA.

Theorem 1.2 ([12]). *Let L be a language, it exists an NFA $\mathcal{A}$ with $L(\mathcal{A}) = L \iff$ it exists a DFA $\mathcal{B}$ with $L(\mathcal{B}) = L$.*
Then, by Theorem 1.1, L is a regular language $\iff$ it exists an NFA $\mathcal{A}$ with $L(\mathcal{A}) = L$.

This theorem means that, for any NFA $\mathcal{A}$, it exists a construction of a DFA $\mathcal{B}$ accepting the same language : $L(\mathcal{A}) = L(\mathcal{B})$. Moreover, if $\mathcal{A}$ has n states, we can construct a DFA $\mathcal{B}$ that will have at most 2^n states. A construction of a DFA $\mathcal{B}$ from an NFA $\mathcal{A}$ is given in Appendix A.

Since DFAs and NFAs accept the same class of languages, we may refer to either of them as *finite automata* in what follows.

1.3 Context Free Grammar

This section explains the basis of notions and notations of grammar theory. Like the previous section, most of the content of this section comes from the book *Introduction to Automata Theory, Language and Computation* [12].

1.3.1 Context Free Grammar

We previously said that there exist languages that cannot be represented as a regular language and, by Theorem 1.1, there exists no DFA accepting these languages. Some examples of these languages are the set of all palindromes over some alphabet and the set of well-parenthesized words.

A *context free grammar* is a way to represent certain languages – some of which are non-regular – over an alphabet Σ. The symbols in Σ are called terminal symbols, or *terminals*. A context free grammar is described with a set of nonterminal symbols, called *variables*. Each variable represents a language and is represented by words over the union of variables and terminals. The description of variables is called the set of *productions*.

Definition 1.7: Context Free Grammar

A *Context Free Grammar* (CFG) is a tuple $\mathcal{G} = (V, T, P, S)$, where :

- V is the set of variables ;
- T is the set of terminals ;
- P is the set of productions. Each production is of the form $A \to \alpha$, where $A \in V$ and α is a word composed of terminals and variables: $\alpha \in (V \cup T)^*$;
- $S \in V$ is the start symbol. It can also be called the *axiom* of $\mathcal{G}$.

By convention, we use capital letters to denote variables.

A CFG works with *derivations*. The derivation of a CFG is the way to obtain words by using productions. A derivation is of the form :

$$\beta_1 A \beta_2 \Rightarrow \beta_1 \alpha \beta_2,$$

with $\beta_1, \beta_2, \alpha \in (V \cup T)^*$, $A \in V$ and $A \to \alpha$ a production of $\mathcal{G}$. We can use $\overset{*}{\Rightarrow}$ as the succession of derivations (note zero or one derivation can also be represented by $\overset{*}{\Rightarrow}$).

By doing successive derivations, we can obtain a word $w \in T^*$. We can define the set of all words that can be obtained with derivations of a CFG $\mathcal{G}$. The obtained set of words is called the *language* of $\mathcal{G}$.

Definition 1.8: Context Free Language

Let $A \in V$ be a variable of a grammar $\mathcal{G} = (V,T,P,S)$. The language of A, noted $L(A)$ is the sets of words $w \in T^*$ that can be formed by succession of derivations of the symbol A with productions of P :

$$L(A) = \left\{ w \in T^* | A \overset{*}{\Rightarrow} w \right\}.$$

Let $\mathcal{G} = (V,T,P,S)$ be a Context Free Grammar. The language of $\mathcal{G}$, called a *Context Free Language* (CFL) and noted $L(\mathcal{G})$, is the language of its axiom $L(S)$:

$$L(\mathcal{G}) = L(S) = \left\{ w \in T^* | S \overset{*}{\Rightarrow} w \right\}.$$

As an example, we describe the CFG $\mathcal{G}$ over the alphabet $\Sigma = \{a,b\}$. The alphabet Σ is used as the set of terminals, $T = \Sigma = \{a,b\}$. We use only one variable S to construct $\mathcal{G}$, $V = \{S\}$. We define $\mathcal{G} = (\{S\},\Sigma,P,S)$ a CFG, where P is the set of productions defined by :

$$\begin{aligned} S &\rightarrow aSb, \\ S &\rightarrow \varepsilon. \end{aligned}$$

By applying derivations, we can construct the following words :

- $S \overset{*}{\Rightarrow} \varepsilon : S \Rightarrow \varepsilon$;
- $S \overset{*}{\Rightarrow} ab : S \Rightarrow aSb \Rightarrow a \cdot \varepsilon \cdot b = ab$:
- $S \overset{*}{\Rightarrow} a^3b^3 : S \Rightarrow aSb \Rightarrow a \cdot aSb \cdot b \Rightarrow aa \cdot aSb \cdot bb \Rightarrow aaa \cdot \varepsilon \cdot bbb = a^3b^3$;

We can show that the language $L(\mathcal{G})$ is the nonregular language $\{a^n b^n | n \in \mathbb{N}\}$.

Note that all regular languages are context free languages, but there exist CFLs that are not regular, as we have seen with the previous example. Other CFLs that are nonregular are the language of palindromes over some alphabet and the language of well-parenthesized words.

Note that there exist languages that are not a CFL. However, this work does not focus on any of these languages. An example of a language over an alphabet Σ that is not a CFL is the language $L = \{w \cdot w | w \in \Sigma^*\}$ [12].

1.3.2 Extended Context Free Grammar

In the sequel of the work, we need the definition of *extended* CFG to represent an *abstract JSON schema* (see Section 1.4.3). In an extended CFG, the right-hand side of the productions are regular expressions over the symbols $V \cup T$ instead of just a word.

Definition 1.9: Extended Context Free Grammar

An *Extended Context Free Grammar* (extended CFG) is a tuple $\mathcal{G} = (V, T, P, S)$, where :

- V is the set of variables ;
- T is the set of terminals ;
- P is the set of production. Each production is of the form $A \to \alpha$, where $A \in V$ and α is a *regular expression* (see Definition 1.4) over the symbols $V \cup T$;
- $S \in V$ is the start symbol.

An extended CFG also works with derivations. With extended CFG, when we derive a variable with a production $A \to \alpha$, we derive it by using a word v in the language denoted by the regular expression α :

$$u_1 A u_2 \Rightarrow u_1 v u_2,$$

with $u_1, u_2 \in (V \cup T)^*$, $A \in V$ and $v \in (V \cup T)^*$ a word in the language denoted by the regular expression α. Similarly to the CFG, we use the symbol $\overset{*}{\Rightarrow}$ as the succession of derivations.

The language represented by an extended CFG $\mathcal{G} = (V, T, P, S)$ is noted $L(\mathcal{G})$:

$$L(\mathcal{G}) = \left\{ w \in T^* | S \overset{*}{\Rightarrow} w \right\}.$$

As an example, let $\mathcal{G} = (\{S, R\}, \{a, b, z\}, P, S)$ an extended CFG, where P is the set of the following productions :

$$\begin{aligned} S &\to a(R+S)^* z \\ R &\to abz. \end{aligned}$$

We can show that the language $L(\mathcal{G})$ is nonregular. Here are some examples of words in $L(\mathcal{G})$:

- $S \overset{*}{\Rightarrow} aabzz$: $S \Rightarrow aRz \Rightarrow a \cdot abz \cdot z = aabzz$.
- $S \overset{*}{\Rightarrow} aabzazz$: $S \Rightarrow aRSz \Rightarrow a \cdot abz \cdot Sz \Rightarrow aabz \cdot az \cdot z = aabzazz$.

When we compare the class of languages described by CFGs and extended CFGs, we can show that it describes exactly the same class of languages (proof in Appendix B), the *Context Free Languages* CFL. This means that, any extended CFG can be transform into a *classic* CFG. However, an extended CFG is useful to describe a CFL as it usually uses fewer variables and productions.

1.4 JSON Document and JSON Schema

This section formally defines and describes the JSON documents and the JSON Schemas, and uses some notations and shortcuts from different articles [7,8,16]. Some details are omitted for readability, and we refer to the official JSON website [14] and JSON Schema website [13] for a full description.

1.4.1 Formalism of JSON Document

A JSON (*JavaScript Object Notation*) document is a widely used data format for data exchange and storage, particularly in web applications, Internet of Things... It is designed to be easy to read and write for developers and to be efficiently processed by machines.

Six types of values are possible in a JSON document :

- The `null` value.
- Boolean values `true` or `false`.
- Any positive or negative decimal number is called a `number` value. In particular, a number that is an integer can be called an `integer` value.
- A finite sequence of characters starting and ending with a double quote (") is a `string` value.
- An *object* is an *unordered* set of key-value pairs :

  $$\{k_1 : v_1, k_2 : v_2, ..., k_n : v_n\},$$

  where $k_1, k_2, ...k_n$ are distinct strings called *keys* and $v_1, v_2, ..., v_n$ are any JSON values.

  An object begins with a left brace ({) and ends with a right brace (}). A key is followed by a colon (:) then by its value. Each key-value pair is separated by a comma (,). As an object is unordered, the objects $\{k_1 : v_1, k_2 : v_2\}$ and $\{k_2 : v_2, k_1 : v_1\}$ are considered as the same object.
- An *array* is an *ordered* collection of values and begins with a left bracket ([) and ends with a right bracket (]) :

  $$[v_1, v_2, ..., v_n].$$

The `null`, `true`, `false`, `number`, `integer` and `string` values are called the *primitive values* of a JSON document.

In this work, we suppose that a JSON document is always an object. With this assumption, we can construct an extended CFG that formally defines all JSON documents. In this CFG, *Object*, *Array* and *Value* are the set of variables, the primitive values and the special characters (comma, colon, braces and brackets) are the set of terminals, and the productions are defined by :

$$
\begin{aligned}
&Object \to \{\} \\
&Object \to \{\texttt{string} : Value(,\texttt{string} : Value)^*\} \\
&Array \to [] \\
&Array \to [Value(,Value)^*] \\
&Value \to \texttt{null} + \texttt{true} + \texttt{false} + \texttt{number} + \texttt{integer} + \texttt{string} + Object + Array.
\end{aligned}
$$

Figure 1.3 shows an example of a JSON Document. We can see that this document is an object containing three keys : "sensor", "status" and "alerts". The values corresponding to these keys are respectively a string, an object and an array.

```
{
  "sensor": "kitchen",
  "status": {
    "temperature": 22.5,
    "humidity": 55
  },
  "alerts": [
     "Warning : Temperature slightly above normal range",
     "Info : Battery level sufficient"
  ]
}
```

Figure 1.3: Example of a JSON Document.

1.4.2 Formalism of JSON Schema

In many scenarios, it can be useful to define a format for the JSON document. For instance, a call to an API may require a specific format to avoid unintended effects. An integrity layer is thus added to validate the JSON document. That's where the *JSON schema* comes in.

A JSON schema is a simple schema language that allows to constrain the structure of JSON documents and provides a framework for verifying the integrity of them. We say that a JSON document *satisfies* the JSON schema if it verifies the constraints imposed by it. The JSON schema will itself be written as a JSON object.

A JSON schema uses several keywords to constraint the set of JSON documents that it specifies. Here are some examples of constraints a JSON schema can do :

- It can impose a string value to respect some regular expression ;
- It can impose a number value to belong to some interval or to be a multiple of some number ;
- It can impose restrictions on key-value pairs of a schema. For example, it imposes which keys are allowed, which keys are required, or it imposes the value paired with some key to satisfy some other schema.

- It can impose restrictions on values present in an array. For example, the values have to respect some schema, or it constrains the array to have a maximum/minimum size.
- Schemas can be combined using boolean operations (disjunction, conjunction and negation). For instance, a value may need to match either one schema or another, or be explicitly excluded by a given schema.
- A schema can use a recursive structure to define the format of a JSON document.

Figure 1.4 describes a JSON schema. The structure of the schema constrains the document to be an object with three required keys : `"sensor"`, `"status"` and `"alerts"`. The values paired with these keys must respectively be a string, an array of string and an object also described by the schema. We can see that the previous example of JSON document in Figure 1.3 satisfies this JSON schema.

```
{
    "type": "object",
    "required": ["sensor", "status", "alerts"],
    "properties": {
        "sensor": { "type": "string" },
        "alerts": {
            "type": "array",
            "items": { "type": "string" }
        },
        "status": {
            "type": "object",
            "properties": {
                "temperature": { "type": "number" },
                "humidity": { "type": "integer" }
            }
        }
    }
}
```

Figure 1.4: Example of a JSON Schema.

1.4.3 Abstract JSON Document and Schema

For the purpose of this work, we use abstract JSON values, documents and schemas. These abstractions help us consider JSON documents as words over some alphabet, and JSON schemas as grammars that describe JSON documents. We take the same notation as [8] to describe the abstractions.

Abstract JSON Values and Documents

First, we only consider the type of the primitive values. We abstract all strings to the symbol `s`, all numbers to the symbol `n` and all integers to the symbol `i`. With these abstractions, we define the abstract alphabet of primitive values :

$$\Sigma_{pVal} = \{\texttt{null}, \texttt{true}, \texttt{false}, \texttt{s}, \texttt{n}, \texttt{i}\}.$$

Then, for objects and arrays, to avoid any confusion with mathematical symbols, we replace the coma (,) by the symbol #, the left brace ({) and right brace (}) respectively by the symbols $\langle$ and $\rangle$, and the left bracket ([) and right bracket (]) respectively by the symbols $\sqsubset$ and $\sqsupset$. Furthermore, the colon (:) between each key-value pair is omitted.

Finally, we consider each keys as a unique symbol. We write Σ_{key} the finite alphabet of allowed keys in a JSON document.

With these abstractions, we denote Σ_{JSON} the alphabet of JSON documents :

$$\Sigma_{JSON} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#, \langle, \rangle, \sqsubset, \sqsupset\}.$$

As an example, the JSON document given in Figure 1.3 is abstracted as the word :

$$\langle\ \texttt{sensor s \# status} \langle\ \texttt{temperature n \# humidity i}\ \rangle\ \texttt{\# alerts} \sqsubset\ \texttt{s \# s}\ \sqsupset\ \rangle,$$

with $\Sigma_{key} = \{\texttt{sensor}, \texttt{status}, \texttt{alerts}, \texttt{temperature}, \texttt{humidity}\}$. Note that we usually use the symbols $k_1, k_2, ..., k_m$ to represent Σ_{key}.

Abstract JSON Schema

We use extended context-free grammar (see Definition 1.9) and the abstractions mentioned previously to define an *abstract JSON schema*. An abstract JSON schema is represented by an extended CFG $\mathcal{G}$ that uses symbols of Σ_{JSON} as the set of terminals and $\mathcal{S} = \{S_0, S_1, ..., S_n\}$ as the set of variables, with S_0 the start symbol. The productions of $\mathcal{G}$ are of the form :

- **Primitive Schema :** $S \to v$, with $v \in \Sigma_{pVal}$;
- **Object Schema :** $S \to \langle k_1 S_1 \# ... \# k_m S_m \rangle$, with $m \geq 0$ and pairwise distinct keys $k_i \in \Sigma_{key}$ for all $i \in \{1, ..., m\}$;
- **Array Schema :** $S \to \sqsubset\ \varepsilon + (S_i(\#S_i)^*)\ \sqsupset$, or $S \to \sqsubset\ S_i \# ... \# S_i\ \sqsupset$ for some occurrence of S_i ;
- **Disjunction Operations :** $S \to S_1 + ... + S_m$, with $m \geq 0$.

Note that, in the definition of JSON schema (see Section 1.4.2), we mentioned that a schema could use any boolean operations, including negation and conjunction. However, one can show that these boolean operations over a JSON schema can be transformed to obtain a set of productions using regular expressions [8].

As an object of a JSON document is unordered, that is it may have its keys in any order, the extended CFG $\mathcal{G}$ must be *closed* over object schemas. This means that, whenever it contains a production $S \to \langle k_1 S_1 \# ... \# k_m S_m \rangle$, it also contains all production $S \to \langle k_{i_1} S_{i_1} \# ... \# k_{i_m} S_{i_m} \rangle$, where $(i_1, ..., i_m)$ is any permutation of $(1, ..., m)$.

If we take the JSON schema given as example in Figure 1.4, we can write the extended CFG $\mathcal{G} = (\mathcal{S}, \Sigma_{JSON}, P, S_0)$ abstracting this schema, with the set of productions P :

$$
\begin{aligned}
&S_0 \rightarrow \langle \texttt{sensor } S_1\texttt{\#alerts } S_2\texttt{\#status } S_3 \rangle \\
&S_1 \rightarrow \texttt{s} \\
&S_2 \rightarrow \sqsubset \varepsilon + (S_1(\#S_1)^*) \sqsupset \\
&S_3 \rightarrow \langle \texttt{temperature } S_4\texttt{\#humidity } S_5 \rangle \\
&S_3 \rightarrow \langle \texttt{temperature } S_4 \rangle \\
&S_3 \rightarrow \langle \texttt{humidity } S_5 \rangle \\
&S_3 \rightarrow \langle \varepsilon \rangle \\
&S_4 \rightarrow \texttt{n} \\
&S_5 \rightarrow \texttt{i}
\end{aligned}
$$

where we add to S_0 and S_3 all the productions mandatory to *close* the extended CFG $\mathcal{G}$. We usually omit to explicitly close the extended CFG to improve readability.

The previous definition of an extended CFG representing a JSON schema is useful, as it's easy to show that any JSON schema can be represented as an extended CFG $\mathcal{G}$ of this form, such that any abstracted JSON document that satisfies the schema is in $L(\mathcal{G})$ [8]. However, in this work, we will need another form for a CFG describing a JSON schema.

Theorem 1.3 ([8])**.** *Let $\mathcal{G} = (\mathcal{S}, \Sigma_{JSON}, P, S_0)$ be an extended CFG defining an abstract JSON schema object. We can construct an extended CFG $\mathcal{G}' = (\mathcal{S}', \Sigma_{JSON}, P', S'_0)$ such that :*

- *The left-hand sides of productions are pairwise distinct ;*
- *Each production is of the form $S'_i \rightarrow a_i e_i \overline{a_i}$ with $a_i \in \{\langle, \sqsubset\}$, $\overline{a_i} = \rangle$ (resp. $\sqsupset$) if $a_i = \langle$ (resp. $\sqsubset$) and e_i is a regular expression over $\mathcal{S}' \cup \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#\}$;*

and $L(\mathcal{G}) = L(\mathcal{G}')$.

Note that the initial theorem explained in the article [8] says that there can be more than one axiom. Since all axioms of a JSON document are an object schema, we can create a new variable S'_0 which is represented by the production $S'_0 \rightarrow \langle e_0 \rangle$, with α_0 a disjunction of the regular expressions of all the previous axioms.

As an example, you can see in Appendix C the transformation of the extended CFG that abstracts the JSON schema described in Figure 1.4.

Chapter 2

Visibly System of Procedural Automata

A *System of Procedural Automata* (SPA) is an extension of DFAs that mutually call each other [10]. SPAs provide a mechanism for accepting any Context Free Grammar (see Definition 1.7), under the condition that each right-hand side of the productions begins with a unique call symbol (specific for the production) and ends with a return symbol (unique for the SPA). Although there exists a transformation that converts any CFG into this form, it is usually impractical to use, as it becomes hard to assess if a word belongs to the language of the CFG after this transformation.

We propose a new kind of SPA, that we call *Visibly System of Procedural Automata* (VSPA). VSPAs are built in such a way that it is easy for a VSPA to accept a CFL where each right-hand side of the productions starts with a call symbol (from a designated *call alphabet*, defined with the VSPA) and ends with a return symbol (from a designated *return alphabet*, defined with the VSPA). More specifically, we built VSPAs such that any JSON schema can be the language of a VSPA.

In this chapter, we first introduce the formalism of VSPAs, describe their behavior while reading a word, and outline some properties of them. We then prove that any JSON schema can be represented by a VSPA.

2.1 Visibly System of Procedural Automata – VSPA

2.1.1 Formalism of VSPA

Just like automata (see Definition 1.1), VSPAs are designed to accept words that belong to a specific language. These words are over a specific alphabet, denoted by Σ. However, unlike automata, when a VSPA reads a symbol of the word, its behavior may vary, depending on what kind of symbols it reads.

We divide the alphabet Σ into three distinct subsets. The first one is called the *internal alphabet*, which contains all the symbols for which the VSPA behaves like an NFA (see Definition 1.5). The two other alphabets are the *call alphabet* and the *return alphabet*. We explain later how the VSPA behaves when it reads a *call symbol* or a *return symbol*.

In our case, we use a specific return alphabet such that each return symbol corresponds uniquely to a call symbol, and vice versa. For a call symbol a, we denote its corresponding return symbol by $\overline{a}$. That is inspired by well-parenthesized words, where a closing parenthesis must match an opening

parenthesis. For instance, the symbol "$\overline{(}$" corresponds to the close parenthesis ")". JSON documents have a similar case as opening and closing parentheses with the left and right braces and brackets.

A VSPA is a set of NFAs that mutually call each other. Each NFA that composes the VSPA is denoted by a unique *procedural symbol*. We call *procedural alphabet* the set of all procedural symbols.

The *VSPA alphabet* is the union of the internal, call, return and procedural alphabets. Note that these four alphabets are disjoint, meaning that no symbol can belong to more than one of these alphabets.

Definition 2.1: VSPA Alphabet

A *VSPA alphabet*, noted $\widehat{\Sigma}$, is the union of the following disjoint alphabets :

$$\widehat{\Sigma} = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{proc},$$

with :

- Σ_{int} the *internal alphabet*, the set of *internal symbols* ;
- Σ_{call} the *call alphabet*, the set of *call symbols* ;
- Σ_{ret} the *return alphabet*, the set of *return symbols* ;
- Σ_{proc} the *procedural alphabet*, the set of *procedural symbols* ;

We use a particular return alphabet, where each return symbol corresponds to a unique call symbol :

$$\Sigma_{ret} = \{\overline{a} | a \in \Sigma_{call}\}$$

By convention, we use capital letters to represent symbols of Σ_{proc} (as it's done with variables in CFGs).

Note that a VSPA is designed to accept words over the language $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$ only. A word accepted by a VSPA will never contain a symbol from the alphabet Σ_{proc}, that is, a VSPA never explicitly reads a procedural symbol.

Each procedural symbol corresponds to a unique NFA that compose the VSPA. We call these NFAs the *procedural automata*.

The transitions of these automata are defined only for symbols belonging to the internal alphabet and the procedural alphabet. We note $\widetilde{\Sigma}$ the union of these two alphabets : $\widetilde{\Sigma} = \Sigma_{int} \cup \Sigma_{proc}$.

Definition 2.2: Procedural Automaton

Let $\widetilde{\Sigma} = \Sigma_{int} \cup \Sigma_{proc}$, a *Procedural Automaton* is a finite automaton (see Definition 1.5) $\mathcal{A}^J = (Q^J, q_0^J, \widetilde{\Sigma}, F^J, \delta^J)$, with $J \in \Sigma_{proc}$ the procedural symbol corresponding to $\mathcal{A}^J$.
We note $\widetilde{L}(\mathcal{A}^J) \subseteq \widetilde{\Sigma}^*$ the regular language (see Definition 1.4) accepted by $\mathcal{A}^J$.

The language $\widetilde{L}(\mathcal{A}^J) \subseteq \widetilde{\Sigma}^*$ is the *regular language* accepted by the procedural automaton $\mathcal{A}^J$ when it behaves as a classical finite automaton. Note that this language is over the alphabet $\widetilde{\Sigma} = \Sigma_{int} \cup \Sigma_{proc}$, when words accepted by a VSPA are over the alphabet $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$.

A *Visibly System of Procedural Automata* is a set of procedural automata that may call each other. When the VSPA reads a call symbol of a word, it must call a procedural automaton. To determine which procedural automata may be called, we define a function $f : \Sigma_{proc} \to \Sigma_{call}$. This function maps any procedural symbol (unique for a procedural automaton) to a call symbol : $f(J) = a$, with $J \in \Sigma_{proc}$ and $a \in \Sigma_{call}$. Note that the same call symbol may be associated with multiple procedural symbols – that is, the function f is surjective.

As a first illustrative example, we define the VSPA alphabet $\widehat{\Sigma} = \{b\} \cup \{a\} \cup \{\overline{a}\} \cup \{R, S\}$, with $\Sigma_{int} = \{b\}$, $\Sigma_{call} = \{a\}$, $\Sigma_{ret} = \{\overline{a}\}$, and $\Sigma_{proc} = \{R, S\}$. We take the procedural automata $\mathcal{A}^S$ and $\mathcal{A}^R$ from Figure 2.1, and we associate the procedural symbol R with the call symbol a : $f(R) = a$.

Suppose that we are in the state q_0^S and that we want to read the word $w = ab\overline{a}$. The first symbol to read is the call symbol a. Therefore, we need to call a procedural automaton. From this state, we see that there exists a transition reading the procedural symbol R. Since $f(R) = a$, and there exists a transition reading R from this state, we can call the procedural automaton $\mathcal{A}^R$, and we move in its initial state q_0^R.

We then need to read the internal symbol b. Since it's an internal symbol, the VSPA simply behaves like an NFA, and follows the transition $q_0^R \xrightarrow{b} q_1^R$.

We finally read the return symbol $\overline{a}$. We must return to the procedural automaton that called $\mathcal{A}^R$, in the state where it was, in our case, q_0^S, and we perform the transition reading R to move in the state q_1^S. Note that if we weren't in a final state (in our case $q_1^R \in F^R$), the word would be rejected.

The run of the word $ab\overline{a}$ as explained above is illustrated in Figure 2.2. This defines how a VSPA works when it reads a call or a return symbol.

Figure 2.1: Example of Procedural Automata.

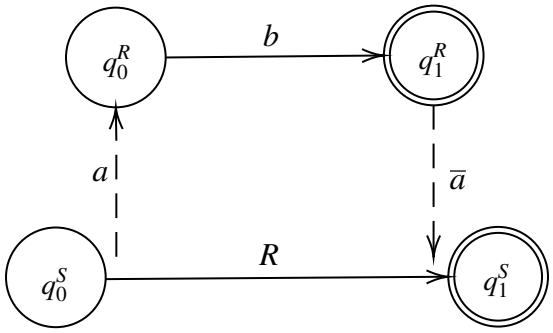


Figure 2.2: Illustration of the run of the word $ab\overline{a}$ with procedural automaton in Figure 2.1.

In order to remember which procedural automaton called another, the VSPA implements a stack. When it reads a call symbol, the VSPA pushes the ongoing state on top of the stack. When it reads a return symbol, the VSPA pops the state from the top of the stack, and moves in this state.

If we take the above example, when we read the call symbol a, we push the state q_1^S on top of the stack. Then, when we read the return symbol $\overline{a}$, we pop q_1^S from the top of the stack, so we know that we have to move in this state.

Note that, in the context of VSPAs, we use a *stack word* over a specific *stack alphabet*. As we can stack any state, the stack alphabet is composed of all the states (which are here considered as a symbol) of all the procedural automata. The top of the stack is considered as the first symbol of the word. An empty stack is the empty word ε.

Definition 2.3: Visibly System of Procedural Automata

A *Visibly System of Procedural Automata*, noted VSPA, is a tuple $\mathcal{A} = \left(\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S\right)$, with :

- $\widehat{\Sigma} = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{proc}$, a VSPA alphabet (see Definition 2.1) ;
- $\Gamma = \left\{q \in Q^J | J \in \Sigma_{proc}\right\}$, the *stack alphabet*, with Q^J the set of states of the procedural automaton $\mathcal{A}^J$;
- $\mathcal{P} = \left\{\mathcal{A}^J | J \in \Sigma_{proc}\right\}$, the set of procedural automata (see Definition 2.2) ;
- $f : \Sigma_{proc} \to \Sigma_{call} : f(J) = a$, a function linking all procedural symbols $J \in \Sigma_{proc}$ to a call symbol $a \in \Sigma_{call}$;
- $S \in \Sigma_{proc}$, the symbol corresponding to the *starting procedural automaton* $\mathcal{A}^S$.

During the processing of a word, the VSPA updates both its current state and its current stack. We refer to the pair of current state and stack as a *configuration*. We write (q^J, γ) the configuration of the VSPA where $q^J \in Q^J$ is the current state, and $\gamma \in \Gamma^*$ the current stack word.

When the VSPA reads a symbol in the alphabet $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$, it updates its configuration – that is, its current state and stack – according to transition rules, called its *semantics*. In the case of reading an internal symbol, the VSPA behaves like an NFA, and updates only its state. However, when it reads a call or a return symbol, both the state and the stack changes.

Note that if no rule can be satisfied according to the semantics, the word is considered invalid, and is therefore rejected.

Definition 2.4: Semantics of a VSPA

The *Semantics* of a VSPA $\mathcal{A} = \left(\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S\right)$ defines its behavior when it reads a symbol $a \in \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$ from a configuration $\left(q^J, \gamma\right)$, with $q \in Q^J$, $J \in \Sigma_{proc}$ and $\gamma \in \Gamma^*$. We can split the semantics into three parts :

- *Interior Rules* ($a \in \Sigma_{int}$):
$$(q^J, \gamma) \xrightarrow{a} (p^J, \gamma),$$
with $p^J \in Q^J$ such that $p^J \in \delta^J(q^J, a)$;
- *Call Rules* ($a \in \Sigma_{call}$):
$$(q^J, \gamma) \xrightarrow{a} (q_0^K, p^J \cdot \gamma),$$
with :
  - $K \in \Sigma_{proc}$ such that $f(K) = a$;
  - $p^J \in Q^J$ such that $p^J \in \delta^J(q^J, K)$;
  - q_0^K, the initial state of the procedural automaton $\mathcal{A}^K$;
- *Return Rules* ($a \in \Sigma_{ret}$):
$$(q^J, p^K \cdot \gamma) \xrightarrow{a} (p^K, \gamma),$$
with :
  - $q^J \in F^J$, a final state of the procedural automaton $\mathcal{A}^J$;
  - $a = \overline{f(J)}$;
  - $p^K \in Q^K$, the state on top of the stack word ;

The first part of the semantics of the VSPA describes its behavior when it reads an internal symbol. As seen before, in that case, the VSPA behaves like a classical NFA : it goes to a reachable state according to the transition function δ^J.

However, we saw that the behavior of a VSPA is different when it reads a call or a return symbol. Suppose the VSPA is in the state q^J and needs to read the word $aw\overline{a}$, with $a \in \Sigma_{call}$ and $w \in \Sigma^*$. Let $K \in \Sigma_{proc}$ a procedural symbol such that $f(K) = a$, and assume there exists the transition $q^J \xrightarrow{K} p^J$. The VSPA first reads the symbol a and *call* the procedural automaton $\mathcal{A}^K$, that is, it goes to the initial state q_0^K. From this state, it reads the word w until it reaches a final state q_F^K of $\mathcal{A}^K$. Then, it reads the return symbol $\overline{a} = \overline{f(K)}$, and finally reaches the state p^J, stored in the stack. We say that $aw\overline{a}$ is a word accepted by $\mathcal{A}^K$, that is $aw\overline{a} \in L(\mathcal{A}^K)$ (see Definition 2.6).

Figure 2.3 illustrates the behavior of the VSPA when it reads a word $aw\overline{a}$.

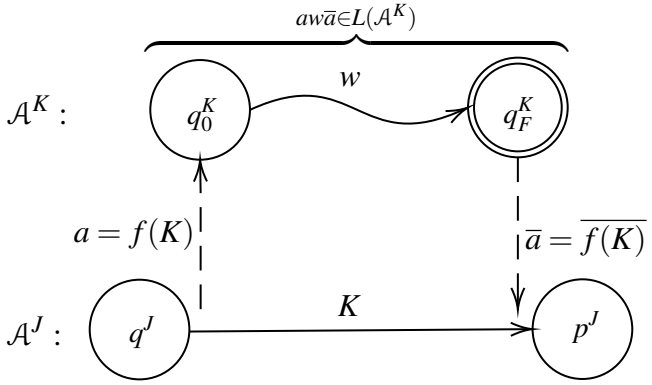


Figure 2.3: Illustration of the call and return rules of a VSPA when it reads a word $aw\overline{a}$.

For more intuition of how a VSPA reads a word, examples are given in Section 2.1.2.

Note that, if all procedural automata $\mathcal{A}^J \in \mathcal{P}$ are deterministic, then the interior and the return rules are deterministic too. However, the call rules may still be nondeterministic, since there may exist two (or more) procedural symbols $K_1, K_2 \in \Sigma_{proc}$ such that $f(K_1) = f(K_2)$ and $\delta^J(q^J, K_1) \neq \delta^J(q^J, K_2)$. The VSPA is deterministic if and only if all its procedural automata are deterministic and if the function f is bijective. We see in the next chapter how the non-determinism is handle by the validation algorithm.

As we did for DFAs and NFAs with the run function (see Definition 1.2), we can define the behavior of a VSPA when reading a word with the *stacked run*. It gives us a reachable configuration when we read a word from an initial configuration.

Definition 2.5: Stacked Run of a VSPA

A *stacked run* of a VSPA $\mathcal{A}$ is a configuration (q', γ') reachable from a starting configuration (q, γ) by reading one by one the symbols of a word $w = a_1 ... a_n$:

$$(q, \gamma) \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (q', \gamma'),$$

$$(q, \gamma) \xrightarrow{w} (q', \gamma').$$

We denote by $\hat{\delta}(q, w, \gamma)$ the set of all the possible stacked runs of the word w from the configurtation (q, γ) :

$$\hat{\delta}(q, w, \gamma) = \left\{ (q', \gamma') \,\middle|\, \exists (q, \gamma) \xrightarrow{w} (q', \gamma') \right\}.$$

A VSPA always begins in the configuration (q_0^S, ε), with q_0^S the initial state of the starting procedural automaton $\mathcal{A}^S$, and ε the empty stack word. We say that a word is *accepted* by the VSPA if it starts by the call symbol $f(S)$ and ends by the return symbol $\overline{f(S)}$ and if, starting from the initial configuration (q_0^S, ε), it exists a stacked run of all other symbols of the word that ends in a configuration (q_F^S, ε), with $q_F^S \in F^S$ a final state of $\mathcal{A}^S$.

Similarly, for any procedural automaton $\mathcal{A}^J$, it accepts a word $aw\overline{a}$ if $a = f(J)$ and if it exists a stacked run $(q_0^J, \varepsilon) \xrightarrow{w} (q_F^J, \varepsilon)$, with q_0^J and q_F^J respectively the initial and a final state of $\mathcal{A}^J$.

Definition 2.6: Language of a VSPA

Let $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S)$ a VSPA and $\mathcal{A}^J \in \mathcal{P}$ a procedural automaton. A word $aw\overline{a}$ is accepted by $\mathcal{A}^J$ if $f(J) = a$ and if it exists a stacked run reading the word w from the configuration (q_0^J, ε) to a configuration (q_F^J, ε), with $q_F^J \in F^J$.
The *Visibly Language of the procedural automaton* $\mathcal{A}^J$, noted $L(\mathcal{A}^J)$, is the set of words accepted by $\mathcal{A}^J$:

$$L(\mathcal{A}^J) = \left\{ f(J) \cdot w \cdot \overline{f(J)} \in \Sigma^* \;\middle|\; \exists q_F^J \in F^J, (q_F^J, \varepsilon) \in \hat{\delta}(q_0^J, w, \varepsilon) \right\}.$$

The *Language of a VSPA*, noted $L(\mathcal{A})$, is the set of words accepted by the starting procedural automaton $\mathcal{A}^S$:

$$L(\mathcal{A}) = L(\mathcal{A}^S) = \left\{ f(S) \cdot w \cdot \overline{f(S)} \in \Sigma^* \;\middle|\; \exists q_F^S \in F^S, (q_F^S, \varepsilon) \in \hat{\delta}(q_0^S, w, \varepsilon) \right\}.$$

Note that we defined here the *visibly language* of a procedural automaton $L(\mathcal{A}^J)$, that is not to be confused with the *regular language* of a procedural automaton $\widetilde{L}(\mathcal{A}^J)$ (see Definition 2.2). The regular language of $\mathcal{A}^J$, over the alphabet $\widetilde{\Sigma} = \Sigma_{int} \cup \Sigma_{proc}$, is its language when it behaves like a classical finite automaton. The visibly language of $\mathcal{A}^J$, over the alphabet $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{return}$, is its language when it behaves like a VSPA.

Further in this work, we refer to $L(\mathcal{A}^J)$ as the *language* of $\mathcal{A}^J$, and we refer to $\widetilde{L}(\mathcal{A}^J)$ as the *regular language* of $\mathcal{A}^J$.

2.1.2 Example of a VSPA

As an example, we use the VSPA alphabet (see Definition 2.1) :

$$\widehat{\Sigma} = \{b\} \cup \{a\} \cup \{z\} \cup \{R, S\},$$

with $\Sigma_{int} = \{b\}$ the internal alphabet, $\Sigma_{call} = \{a\}$ the call alphabet, $\Sigma_{ret} = \{z = \overline{a}\}$ the return alphabet and $\Sigma_{proc} = \{R, S\}$ the procedural alphabet. Figure 2.4 shows two procedural automata (see Definition 2.2) $\mathcal{A}^S$ and $\mathcal{A}^R$ over the alphabet $\widetilde{\Sigma} = \Sigma_{int} \cup \Sigma_{proc} = \{b, R, S\}$.

We construct a VSPA $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S)$, with $\mathcal{P} = \{\mathcal{A}^S, \mathcal{A}^R\}$ the sets of procedural automata, $\Gamma = \{q_0^S, q_0^R, q_1^R\}$ the stack alphabet, S the start symbol and f the function linking procedural symbols to call symbols, with $f(S) = f(R) = a$.

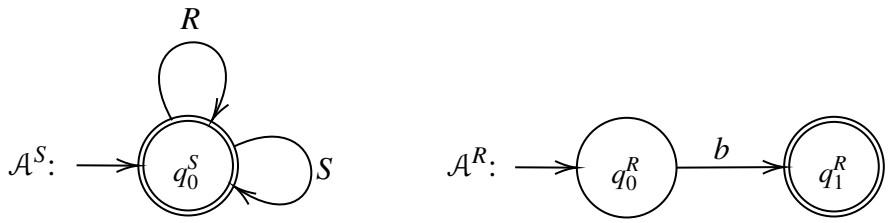


Figure 2.4: Example of VSPA $\mathcal{A}$ composed of two procedural automata $\mathcal{A}^S$ and $\mathcal{A}^R$.

Here are some examples of words accepted by the VSPA $\mathcal{A}$:

- $w_1 = aabzz$:

  To assert that $w_1 \in L(\mathcal{A})$ (see Definition 2.6), we first decompose $w_1 = f(S) \cdot abz \cdot \overline{f(S)}$. Then, we have to verify if it exists a stacked run (see Definition 2.5) such that $(q_0^S, \varepsilon) \xrightarrow{abz} (q_0^S, \varepsilon)$ (because q_0^S is the initial state and the unique final state of $\mathcal{A}^S$).

  We start in the configuration (q_0^S, ε) and we read the call symbol $a \in \Sigma_{call}$. By following the semantics of VSPA (see Definition 2.4), a reachable configuration is (q_0^R, q_0^S), because :

  - $f(R) = a$;
  - $q_0^S \in \delta^S(q_0^S, R)$;
  - q_0^R is the initial state of the procedural automaton $\mathcal{A}^R$.

  From the configuration (q_0^R, q_0^S), we read the internal symbol $b \in \Sigma_{int}$. We can go to the configuration (q_1^R, q_0^S), because $q_1^R \in \delta^R(q_0^R, b)$ is the only reachable state according to the transition function δ^R.

  Finally, we read the return symbol $z \in \Sigma_{ret}$ in the configuration (q_1^R, q_0^S). By following the semantics of VSPAs, because q_1^R is a final state of $\mathcal{A}^R$ and $z = \overline{f(R)}$, we can pop q_0^S from the stack and go to the configuration (q_0^S, ε).

  The full stacked run of abz is therefore :

  $$(q_0^S, \varepsilon) \xrightarrow{a} (q_0^R, q_0^S) \xrightarrow{b} (q_1^R, q_0^S) \xrightarrow{z} (q_0^S, \varepsilon).$$

  Then, we can assess that $w_1 = aabzz = f(S)abz\overline{f(S)} \in L(S) = L(\mathcal{A})$, by Definition 2.6 of languages of VSPAs. The stacked run of the word is illustrated in Figure 2.5

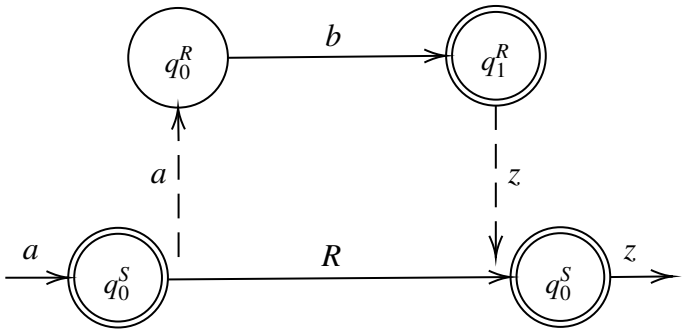


Figure 2.5: Example of an accepting stacked run of the word $w_2 = aabzz$ for the VSPA $\mathcal{A}$ illustrated in Figure 2.4.

- $w_2 = aabzazz$:

  We first assess that $w_2 = f(S)abzaz\overline{f(S)}$. A possible stacked run for the word $abzaz$ starting from the initial configuration (q_0^S, ε) is :

$$(q_0^S, \varepsilon) \xrightarrow{a} (q_0^R, q_0^S) \xrightarrow{b} (q_1^R, q_0^S) \xrightarrow{z} (q_0^S, \varepsilon) \xrightarrow{a} (q_0^S, q_0^S) \xrightarrow{z} (q_0^S, \varepsilon).$$

  Because the previous run exists, w_2 is accepted by the VSPA $\mathcal{A}$: $w_2 \in L(\mathcal{A})$. This run is illustrated in Figure 2.6.

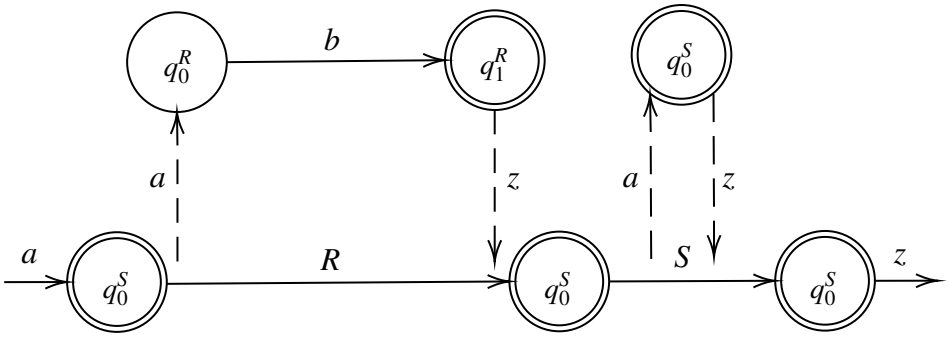


Figure 2.6: Example of an accepting stacked run of the word $w_2 = aabzazz$ for the VSPA $\mathcal{A}$ illustrated in Figure 2.4.

We can show that the language of the VSPA $\mathcal{A}$ is the same as the language of the extended CFG $\mathcal{G} = (\{S,R\}, \{a,b,z\}, P, S)$, where P is the set of the following productions :

$$\begin{aligned} S &\rightarrow a(R+S)^*z \\ R &\rightarrow abz. \end{aligned}$$

Moreover, we can show that the regular languages of the procedural automata $\mathcal{A}^S$ and $\mathcal{A}^R$ are :

- $\widetilde{L}(\mathcal{A}^S) = \Sigma_{proc}^*$, denoted by the regular expression $(R+S)^*$;
- $\widetilde{L}(\mathcal{A}^R) = \{b\}$, denoted by the regular expression b.

2.1.3 Properties of VSPAs

We can define some properties of VSPAs that will help us later in the work, and increase the vision of how VSPAs work. Note that we reuse some existing notations for some of the following definitions [8].

Well-Matched Word and Depth of a Word

To illustrate the first property, we first assume that the call alphabet and the return alphabet are composed of only one symbol. The call symbol can be represented by an open parenthesis "(", and the return symbol by a closed parenthesis ")". Instinctively, if a word over this alphabet is accepted by a VSPA, it must be well-parenthesized, since each call symbol must be paired with a return symbol, and conversely.

We define the notion of *well-matched words*, which formally generalizes the concept of well-parenthesized words to call and return alphabets with more than one symbol. A word is well-matched if every call symbol a is correctly paired with its corresponding return symbol $\overline{a}$, and no unmatched call or return symbol appears between a and $\overline{a}$.

Let's take parentheses and brackets as an example, we define $\Sigma_{call} = \{(, [\}$ and $\Sigma_{ret} = \{),]\}$. It is clear that the word "([])" is well-matched, but the word "([)]" isn't, as the parenthesis is closed before the open bracket inside the parenthesis is.

Definition 2.7: Well-Matched Word

Given an alphabet $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$, with $\Sigma_{ret} = \{\overline{a} | a \in \Sigma_{call}\}$, the set of *Well-Matched Words* over Σ, noted $WM(\Sigma)$, is recursively defined by :

- Let $w \in \Sigma_{int}^*$: $w \in WM(\Sigma)$;
- Let $w \in WM(\Sigma)$ and $a \in \Sigma_{call}$: $aw\overline{a} \in WM(\Sigma)$;
- Let $w, w' \in WM(\Sigma)$: $w \cdot w' \in WM(\Sigma)$.

If we imagine again a parenthesized word, the depth of the word is determined by counting the deepest level of opening parentheses before they are closed. For example, in the word "$a(b(c)d)$", the depth is 2, since the deepest nested part is 'c' inside 2 layers of parentheses. A word with no parenthesis has a depth of 0. In the case of a well-matched word, the depth is the deepest level of call symbols that haven't matched their return symbols yet.

Before formally defining the depth, we first define the *call-return balance*, which defines the number of unmatched call symbols of a word. This definition will help us to define the depth.

Definition 2.8: Call-Return Balance

The *Call-Return Balance* is a function $\beta : \Sigma \to \mathbb{Z}$ recursively defined as :

$$\beta(a \cdot u) = \beta(u) + \begin{cases} 1 & \text{if } a \in \Sigma_{call} \\ -1 & \text{if } a \in \Sigma_{ret} \\ 0 & \text{if } a \in \Sigma_{int} \end{cases},$$

and $\beta(\varepsilon) = 0$.

With the call-return balance, we can define some properties of well-matched words.

Let $w \in WM(\Sigma)$, we can prove the properties :

$$\beta(w) = 0, \tag{2.1}$$

$$\forall u \in Pref(w) : \beta(u) \geq 0, \tag{2.2}$$

$$\forall u \in Suff(w) : \beta(u) \leq 0. \tag{2.3}$$

Property (2.1) is trivial. By Definition 2.7 of well-matched words, every call symbol must be matched with its corresponding return symbol. Therefore, there are no unmatched call nor return symbols, and there are as many call and return symbols in a well-matched word.

Properties (2.2) and (2.3) are also evident. Imagine a well-matched word such that it exists a prefix u of it with $\beta(u) < 0$. This means that u contains more return symbol than call symbols. Therefore, there is at least one return symbol that isn't matched with any previous call symbol. Thus, the word w has the same problem. However, by Definition 2.7, a return symbol appears after its matching call symbol. Therefore, w isn't well-matched. A similar reasoning can be done to prove Property (2.3).

Note that, for any word w, if it respects two of the above properties, w respects the third one. Note also that, in the case there is only one symbol in the call and in the return alphabet, these properties are sufficient to prove that w is well-matched.

With the definition of call-return balance, it is now easy to formally define the depth of a well-matched word. The depth is the largest number of unmatched call symbols at any point in the word.

Definition 2.9: Depth of a Word

Let $w \in \Sigma$, with Σ a VSPA Alphabet. The depth of w, noted $depth(w)$ is the maximal number of unmatched call symbols among the prefixes of w :

$$depth(w) = \max_{u \in Pref(w)} \beta(u).$$

Note that, by Property (2.2), it is clear that the $depth$ of a well-matched word is always greater or equal to zero.

In this work, the depth is mainly useful to prove some properties or theorems.

Properties of Words accepted by a VSPA

In this section, we present some properties of words that are accepted by a procedural automaton or a VSPA. The proofs of all these properties can be found in Appendix D.

We consider in this section the VSPA $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S)$ as defined in Definition 2.3.

The first property of a VSPA was already enounced before. For a word w accepted by a procedural automaton $\mathcal{A}^J$, each call symbol must be matched with a return symbol. That is, a word accepted by a procedural automaton must be well-matched.

$$\begin{aligned} w \in L(\mathcal{A}^J) &\Rightarrow w \in WM(\Sigma), \\ L(\mathcal{A}^J) &\subseteq WM(\Sigma). \end{aligned} \tag{2.4}$$

The language of a VSPA is equal to the language of its starting procedural automaton (see Definition 2.6) : $L(\mathcal{A}) = L(\mathcal{A}^S)$. Therefore, a corollary of Property (2.4) is that any word accepted by a VSPA is well-matched :

$$L(\mathcal{A}) \subseteq WM(\Sigma). \tag{2.5}$$

We recall that we note $\widetilde{L}(\mathcal{A}^J)$ the regular language – over the alphabet $\widetilde{\Sigma} = \Sigma_{int} \cup \Sigma_{proc}$ – accepted by the procedural automaton $\mathcal{A}^J$ if it behaves like an NFA (see Definition 2.2). By Definition 1.3, a word $w \in \widetilde{\Sigma}^*$ belongs to the regular language $\widetilde{L}(\mathcal{A}^J)$ if it exists a run $q_0^J \xrightarrow{w} q_F^J$, with q_0^J and q_F^J respectively the initial state and a final state of $\mathcal{A}^J$.

This language is not to be confused with the language $L(\mathcal{A}^J)$ – over the alphabet $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{return}$ – that is accepted by the procedural automaton $\mathcal{A}^J$ if it behaves like a VSPA. By Definition 2.6, a word $aw\overline{a} \in \Sigma^*$ belongs to the language $L(\mathcal{A}^J)$ if $f(J) = a$ and it exists a stacked run $(q_0^J, \varepsilon) \xrightarrow{w} (q_F^J, \varepsilon)$, with q_0^J and q_F^J respectively the initial state and a final state of $\mathcal{A}^J$.

If the word w belongs to the alphabet $\Sigma^* \cap \widetilde{\Sigma}^* = \Sigma_{int}^*$, and belongs to the regular language $\widetilde{L}(\mathcal{A}^J)$, then the word $aw\overline{a}$ – with $f(J) = a$ – must belong to the language $L(\mathcal{A}^J)$, and conversely :

$$\begin{aligned} &\text{Let } w \in \Sigma_{int}^*,\ f(J) = a : \\ &\quad w \in \widetilde{L}(\mathcal{A}^J) \iff aw\overline{a} \in L(\mathcal{A}^J). \end{aligned} \tag{2.6}$$

The next property is a generalization of Property (2.6). Let $w \in \widetilde{\Sigma}$ such that $w \in \widetilde{L}(\mathcal{A}^J)$, that is, it exists a run from the initial state to a final state of $\mathcal{A}^J$. We can write $w = u_0 K_1 u_1 .... u_{m-1} K_m u_m$, with $u_i \in \Sigma_{int}^*$ and $K_i \in \Sigma_{proc}$ for all i (note that some u_i may be equal to ε). Let $w_i \in \Sigma^*$ be a word accepted by the procedural automaton $\mathcal{A}^{K_i}$, $w_i \in L(\mathcal{A}^{K_i})$. Then, for all i, we can replace K_i by w_i in the word w to obtain a word over Σ such that it exists a stacked run of $w' = u_0 w_1 u_1 ... u_{m-1} w_m u_m$ from the initial state to a final state of $\mathcal{A}^J$ (this stacked run is illustrated in figure 2.7) :

$$\begin{aligned} &\text{let } u_i \in \Sigma_{int}^*, K_i \in \Sigma_{proc} \\ &\text{s.t. } u_0 K_1 u_1 .... u_{m-1} K_m u_m \in \widetilde{L}(\mathcal{A}^J) \Rightarrow \\ &\forall w_i \in L(\mathcal{A}^{K_i}) : f(J) u_0 w_1 u_1 ... u_{m-1} w_m u_m \overline{f(J)} \in L(\mathcal{A}^J). \end{aligned} \tag{2.7}$$

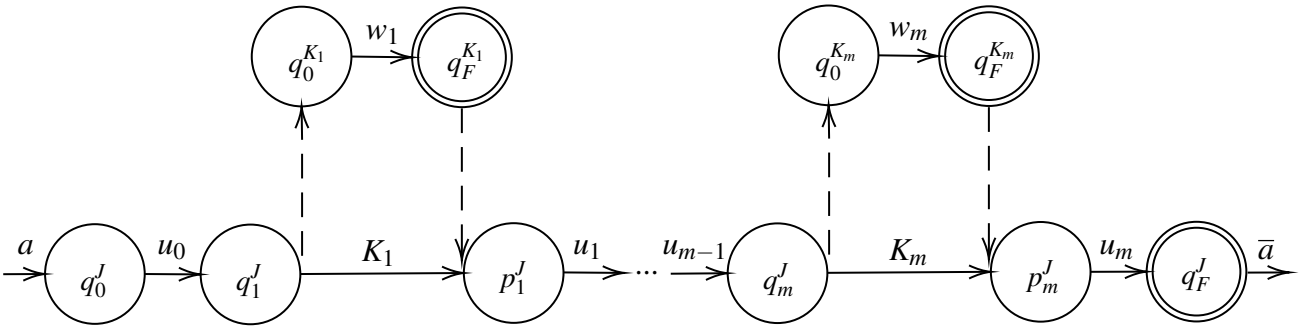


Figure 2.7: Illustration of Property (2.7), run of the word $u_0 K_1 u_1 ... u_{m-1} K_m u_m \in \widetilde{\Sigma}$ and stacked run of the word $a u_0 w_1 u_1 ... u_{m-1} w_m u_m \overline{a} \in \Sigma$.

The mutual statement of Property (2.7) isn't true (see Appendix D). A word $w_i \in L(\mathcal{A}^{K_i})$ could also belong to the language of another procedural automaton, which can make the mutual statement false. We must adapt the statement such that if w_i is accepted by a set of procedural automaton, then it exists one that can replace w_i in the word.

To simplify, if it exists a stacked from a configuration (q, γ) to a configuration (p, γ) that reads the well-matched word $aw\overline{a}$, then there exists a transition between the states q and p that read a procedural symbol J with $aw\overline{a} \in \mathcal{A}^J$:

$$\begin{aligned} &\text{let } w \in WM(\Sigma) \text{ s.t. } \exists (q, \gamma) \xrightarrow{aw\overline{a}} (p, \gamma) \Rightarrow \\ &\exists J \in \Sigma_p \text{ s.t. } aw\overline{a} \in L(\mathcal{A}^J) \text{ and } p \in \delta(q, J). \end{aligned} \tag{2.8}$$

Recall that the proof of all the properties presented in this section can be found in Appendix D.

2.2 VSPA Applied to JSON

2.2.1 Formalism of VSPA with JSON documents and schemas

In this Section, we consider the abstraction described in Section 1.4.3 :

- a JSON document J is a word over the alphabet $\Sigma_{JSON} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#, \langle, \rangle, \sqsubset, \sqsupset\}$;
- a JSON schema is a grammar $\mathcal{G} = (\mathcal{S}, \Sigma_{JSON}, P, S_0)$, with $\mathcal{S} = \{S_0, ..., S_n\}$ as described in Theorem 1.3 :
  - The left-hand sides of productions in P are pairwise distinct ;
  - For all $S_i \in \mathcal{S}$, its production is of the form $S_i \to a_i e_i \overline{a_i}$, with $a_i \in \Sigma_{call}$ and e_i a regular expression over $\Sigma_{int} \cup \mathcal{S}$;
- a JSON document J satisfies the JSON schema $\mathcal{G}$ if and only if $J \in L(\mathcal{G})$.

We create the VSPA Alphabet from the variables and the terminals of a JSON schema grammar :

$$\widehat{\Sigma} = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{proc},$$

with :

- $\Sigma_{int} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#\}$, the *internal* alphabet ;
- $\Sigma_{call} = \{\langle, \sqsubset\}$, the *call* alphabet ;
- $\Sigma_{ret} = \{\rangle, \sqsupset\}$, the *return* alphabet ;
- $\Sigma_{proc} = \mathcal{S} = \{S_0, ...S_n\}$, the *procedural* alphabet.

Moreover, the return alphabet can be written as $\Sigma_{ret} = \{\overline{a} | a \in \Sigma_{call}\}$, with $\overline{a} = \rangle$ (resp. $\sqsupset$) if $a = \langle$ (resp. $\sqsubset$).

The productions of the JSON schema $\mathcal{G}$ are of the form $a_i e_i \overline{a}_i$, with $a_i \in \Sigma_{call}$, $\overline{a}_i \in \Sigma_{ret}$ and e_i a regular expression over $\widetilde{\Sigma} = \Sigma_{int} \cup \mathcal{S}$ (see Theorem 1.3). As e_i is a regular expression, by Theorem 1.1, we can construct a procedural automaton $\mathcal{A}^{S_i}$ over the alphabet $\widetilde{\Sigma}$ such that its regular language $\widetilde{L}(\mathcal{A}^{S_i})$ is denoted by the regular expression e_i.

We create a VSPA $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S^0)$, with :

- $\mathcal{P} = \{\mathcal{A}^{S_i} | S_i \in \mathcal{S}\}$ is the set of all procedural automata, with $\widetilde{L}(\mathcal{A}^{S_i})$ is denoted by the regular expression e_i ;
- $f : \Sigma_{proc} \to \Sigma_{call} : f(S_i) = a_i$;
- $S^0 \in \Sigma_{proc}$ with S^0 the start symbol of the JSON schema $\mathcal{G}$;

We will later show that this VSPA construction is valid and can be applied to any JSON schema.

2.2.2 Example of a VSPA applied to a JSON schema

As an example, we take the JSON schema described in Figure 1.4. This JSON schema can be abstracted to the grammar $\mathcal{G} = (\mathcal{S}, \Sigma_{JSON}, P, S_0)$, with $\mathcal{S} = \{S_0, S_1, S_2\}$ and P the sets of productions :

$$\begin{aligned} S_0 &\to \langle k_1 \mathtt{s} \# k_2 S_1 \# k_3 S_2 \rangle \\ S_1 &\to \sqsubset \varepsilon + (\mathtt{s}(\#\mathtt{s})^*) \sqsupset \\ S_2 &\to \langle (k_4 \mathtt{n} \# k_5 \mathtt{i}) + (k_4 \mathtt{n}) + (k_5 \mathtt{i}) + \varepsilon \rangle \end{aligned}$$

This example doesn't close the grammar over object schema, which means that the keys in an object must respect the order given in the productions. Not closing the grammar will result in fewer states in procedural automata, which leads to more readability. Note that procedural automaton on closed grammar can be constructed.

We create three procedural automata $\mathcal{A}^{S_0}$, $\mathcal{A}^{S_1}$ and $\mathcal{A}^{S_2}$ based on the regular expressions described in the productions of $\mathcal{G}$ (see Figure 2.8). We construct the VSPA with these procedural automata and with the function :

$$f : \Sigma_{proc} \to \Sigma_{call} : \begin{cases} f(S_0) = f(S_2) = \langle \\ f(S_1) = \sqsubset \end{cases} .$$

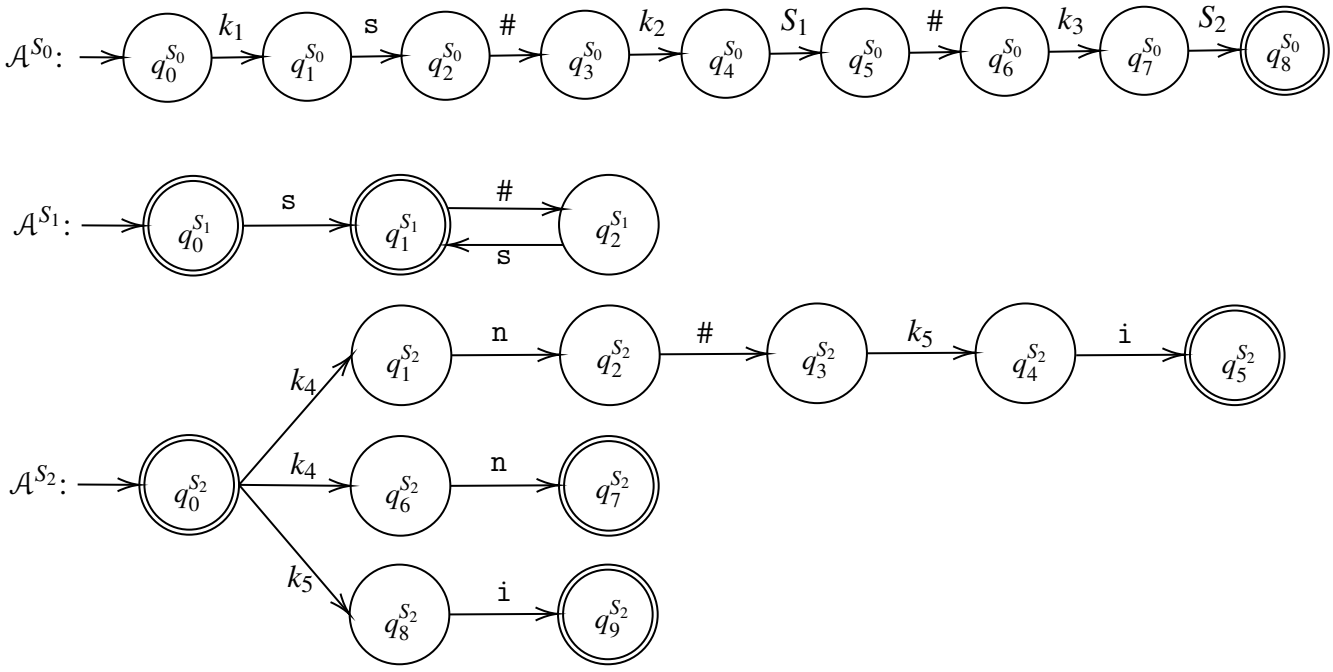


Figure 2.8: Procedural Automata describing the abstract JSON schema in Figure 1.4.

Let $J \in \Sigma_{JSON}$ be the abstract JSON document in Figure 1.3, such that :

$$J = \langle k_1 \mathtt{s} \# k_2 \sqsubset \mathtt{s} \# \mathtt{s} \# \mathtt{s} \sqsupset \# k_3 \langle k_4 \mathtt{s} \# k_5 \mathtt{i} \rangle \rangle.$$

The word J can be written as $J = \langle w \rangle$. The stacked run of the word w is described by :

- The VSPA starts in the configuration $\left(q_0^{S_0}, \varepsilon\right)$. It first reads the word $k_1\mathtt{s}\#k_2$, which is over the internal alphabet Σ_{int}, meaning that there is no stack change. The VSPA behaves like an NFA :

$$\left(q_0^{S_0}, \varepsilon\right) \xrightarrow{k_1} \left(q_1^{S_0}, \varepsilon\right) \xrightarrow{\mathtt{s}} \left(q_2^{S_0}, \varepsilon\right) \xrightarrow{\#} \left(q_3^{S_0}, \varepsilon\right) \xrightarrow{k_2} \left(q_4^{S_0}, \varepsilon\right).$$

- When it reads $\sqsubset$, the VSPA is in the state $q_4^{S_0}$. $\sqsubset$ is a *call* symbol, we follow the semantics of VSPA (see Definition 2.4) : since there is the transition $q_4^{S_0} \xrightarrow{S_1} q_5^{S_0}$, and $f(S_1) = \sqsubset$, it can go to the initial state $q_0^{S_1}$ of the procedural automaton $\mathcal{A}^{S_1}$ and push the state $q_5^{S_0}$ on top of the stack :

$$\left(q_4^{S_0}, \varepsilon\right) \xrightarrow{\sqsubset} \left(q_0^{S_1}, q_5^{S_0}\right).$$

- From the configuration $\left(q_0^{S_1}, q_5^{S_0}\right)$, the VSPA reads the word $\mathtt{s}\#\mathtt{s}\#\mathtt{s}$, with no stack change. It follows the transitions :

$$\left(q_0^{S_1}, q_5^{S_0}\right) \xrightarrow{s} \left(q_1^{S_0}, q_5^{S_0}\right) \xrightarrow{\#} \left(q_2^{S_1}, q_5^{S_0}\right) \xrightarrow{s} \left(q_1^{S_1}, q_5^{S_0}\right) \xrightarrow{\#} \left(q_2^{S_1}, q_5^{S_0}\right) \xrightarrow{s} \left(q_1^{S_1}, q_5^{S_0}\right).$$

- Then, it reads a return symbol $\sqsupset$. As $q_5^{S_1} \in F^{S_1}$ is a final state and $\sqsupset = \overline{f(S_1)}$, it can pop the state $q_5^{S_0}$ from the stack and go to this state :

$$\left(q_5^{S_1}, q_5^{S_0}\right) \xrightarrow{\sqsupset} \left(q_5^{S_0}, \varepsilon\right).$$

- The VSPA then reads $\#k_3$, following the transition :

$$\left(q_5^{S_0}, \varepsilon\right) \xrightarrow{\#} \left(q_6^{S_0}, \varepsilon\right) \xrightarrow{k_3} \left(q_7^{S_0}, \varepsilon\right).$$

- It reads the call symbol $\langle$. As there is the transition $q_7^{S_0} \xrightarrow{S_2} q_8^{S_0}$, and $f(S_2) = \langle$, it can go to the initial state $q_0^{S_2}$ of the procedural automaton $\mathcal{A}^{S_2}$:

$$\left(q_7^{S_0}, \varepsilon\right) \xrightarrow{\langle} \left(q_0^{S_2}, q_8^{S_0}\right).$$

- The VSPA reads the word $k_4\mathtt{n}\#k_5\mathtt{i}$. A possible stacked run is the following :

$$\left(q_0^{S_2}, q_8^{S_0}\right) \xrightarrow{k_4} \left(q_1^{S_2}, q_8^{S_0}\right) \xrightarrow{\mathtt{n}} \left(q_2^{S_2}, q_8^{S_0}\right) \xrightarrow{\#} \left(q_3^{S_2}, q_8^{S_0}\right) \xrightarrow{k_5} \left(q_4^{S_2}, q_8^{S_0}\right) \xrightarrow{\mathtt{i}} \left(q_5^{S_2}, q_8^{S_0}\right).$$

- Finally, it reads the symbol $\rangle$ from a final state, pops the state $q_8^{S_0}$ from the stack and go in that state :

$$\left(q_1^{S_2}, q_8^{S_0}\right) \xrightarrow{\rangle} \left(q_8^{S_0}, \varepsilon\right).$$

The stacked run of the word w is :

$$\left(q_0^{S_0}, \varepsilon\right) \xrightarrow{w = k_1\mathtt{s}\#k_2\sqsubset\mathtt{s}\#\mathtt{s}\sqsupset\#k_3\langle k_4\mathtt{s}\#k_5\mathtt{i}\rangle} \left(q_8^{S_0}, \varepsilon\right).$$

As $q_0^{S_0}$ and $q_8^{S_0}$ are respectively the initial state and the final state of the starting procedural automaton S_0, the JSON document J respects the condition to be in the language $L(\mathcal{A})$ of the VSPA $\mathcal{A}$:

$$f(S_0) = \langle \text{ and } (q_0^{S_0}, \varepsilon) \xrightarrow{w} (q_8^{S_0}, \varepsilon) \Rightarrow J = \langle w \rangle \in L(\mathcal{A}).$$

2.3 Completeness of VSPA for JSON Schema

The goal of VSPAs is the validation of JSON documents according to a JSON schema $\mathcal{G}$. But, to validate if a JSON document J satisfies a schema $\mathcal{G}$, the VSPA $\mathcal{A}$ needs to have the same language as the JSON schema : $L(\mathcal{A}) = L(\mathcal{G})$. By the way we define the VSPA (see Definition 2.3) and its semantics (see Definition 2.4), we can assess that, for any JSON schema, we can construct a VSPA accepting the same language.

Theorem 2.1 (Completeness of VSPA for JSON Schema)**.** *Let $\mathcal{G}$ be an Extended CFG defining an abstracted JSON schema object. There exists a VSPA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{G})$.*

2.3.1 Proof of Theorem 2.1

We construct a VSPA $\mathcal{A}$ accepting $L(\mathcal{G})$. The steps to construct $\mathcal{A}$ are the following :

1. Theorem 1.3 shows that each JSON schema can be transformed to an extended context free grammar $\mathcal{G}$, with $\mathcal{S} = \{S_0, S_1, ..., S_n\}$ the set of variables (with S_0 the axiom) and Σ_{JSON} the set of terminals. Its productions are left-hand side pairwise distinct and of the form $S_j \to a_j e_i \overline{a_j}$ with $a_j \in \{\langle, \sqsubset\}$ and e_j is a regular expression over $\mathcal{S} \cup \Sigma_{int}$ for all j.

2. We define the VSPA Alphabet (see Definition 2.1) from variables and terminals of $\mathcal{G}$:

$$\widehat{\Sigma} = \Sigma_{JSON} \cup \mathcal{S} = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{proc},$$

   with :

   - $\Sigma_{int} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#\}$, the *internal* alphabet ;
   - $\Sigma_{call} = \{\langle, \sqsubset\}$, the *call* alphabet ;
   - $\Sigma_{ret} = \{\rangle, \sqsupset\}$, the *return* alphabet ;
   - $\Sigma_{proc} = \mathcal{S} = \{S_0, ...S_n\}$, the *procedural* alphabet.

3. As e_j is a regular expression over $\mathcal{S} \cup \Sigma_{int} = \hat{\Sigma}$, we can build a Finite Automaton accepting e_j, for all j (see Theorem 1.1). We call these NFAs $\mathcal{A}^{S_j}$ for all j.

4. We create a VSPA $\mathcal{A} = (\Sigma, \Gamma, \mathcal{P}, f, S_0)$ (see Definition 2.3), with :

   - $\mathcal{P} = \{\mathcal{A}^{S_0}, \mathcal{A}^{S_1}, ..., \mathcal{A}^{S_n}\}$;
   - $f : \Sigma_{proc} \to \Sigma_{call} : f(S_j) = a_j$;
   - $S_0 \in \Sigma_{proc}$ the symbol of the initial procedural automaton, with S_0 the axiom of the grammar $\mathcal{G}$;

We can prove that $w \in L(\mathcal{G}) \Leftrightarrow w \in L(\mathcal{A})$ (meaning that $L(\mathcal{G}) = L(\mathcal{A})$). To prove this, we first prove the following claim :

For any word $aw\overline{a} \in WM(\Sigma_{JSON})$, it holds that, if there is a variable $S_j \in \mathcal{S}$ in grammar $\mathcal{G}$ such that $aw\overline{a} \in L(S_j)$, then the procedural automaton $\mathcal{A}^{S_j}$ accepts this word, $aw\overline{a} \in L(\mathcal{A}^{S_j})$, and conversely :

$$aw\overline{a} \in L(S_j) \iff aw\overline{a} \in L(\mathcal{A}^{S_j}).$$

- $aw\overline{a} \in L(S_j) \Rightarrow aw\overline{a} \in L(\mathcal{A}^{S_j})$:

  As a reminder, the language of a procedural automaton (see Definition 2.6) is defined by :

  $$L(\mathcal{A}^{S_j}) = \left\{ f(S_j) \cdot w \cdot \overline{f(S_j)} \in \Sigma \;\middle|\; \exists p^{S_j} \in F^{S_j}, (p^{S_j}, \varepsilon) \in \hat{\delta}(q_0^{S_j}, w, \varepsilon) \right\}.$$

  To prove this claim, we first observe that $aw\overline{a} = a_j w \overline{a_j}$. The production of S_j is of the form $S_j \to a_j e_j \overline{a_j}$, with e_j a regular expression over $\Sigma_{int} \cup \mathcal{S}$. As $aw\overline{a} \in L(S_j)$, a (resp. $\overline{a}$) must be equal to a_j (resp. $\overline{a_j}$). By *step 4*, we see that $a_j = f(S_j)$.

  To prove the claim, we proceed by induction, using the depth of w (see Definition 2.9).

  Base Case : $depth(w) = 0$.

  As $w \in WM(\Sigma_{JSON})$ and $depth(w) = 0$, it holds that w is a word over the internal alphabet Σ_{int}.

  The procedural automaton $\mathcal{A}^{S_j}$ is constructed to accept the language denoted by the regular expression e_j (see *step 3*). This means that the regular language $\widetilde{L}(\mathcal{A}^{S_j})$ is denoted by the regular expression e_j. Since $w \in \Sigma_{int}^*$ is in the language denoted by e_j, then $w \in \widetilde{L}(\mathcal{A}^{S_j})$.

  Finally, by Property (2.6), it holds that :

  $$w \in \widetilde{L}(\mathcal{A}^{S_j}) \cap \Sigma_{int}^* \Rightarrow f(S_j) w \overline{f(S_j)} = a_j w \overline{a_j} \in L(\mathcal{A}^{S_j}).$$

  Induction Step : $depth(w) > 0$.

  By the hypothesis $a_j w \overline{a_j} \in L(S_j)$ and because of the form of the production of S_j (see Theorem 1.3), it exists the following derivations :

  $$S_j \Rightarrow a_j \cdot u_1 S_{i_1} u_2 S_{i_2} \ldots S_{i_m} u_{m+1} \cdot \overline{a_j} \overset{*}{\Rightarrow} a_j \cdot u_1 \phi_1 u_2 \phi_2 \ldots \phi_m u_{m+1} \cdot \overline{a_j} = a_j w \overline{a_j},$$

  with :

  - $u_k \in \Sigma_{int}^*$ words over internal alphabet, for all $k \in \{1, \ldots, m+1\}$;
  - $S_{i_k} \in \mathcal{S}$ procedural symbols, for all $k \in \{1, \ldots, m\}$;
  - $u_1 S_{i_1} u_2 S_{i_2} \ldots S_{i_m} u_{m+1}$ a word in the language denoted by the regular expression e_j ;
  - $\phi_k = b_k w_k \overline{b_k} \in WM(\Sigma_{JSON})$, with $b_k \in \Sigma_{call}$, such that $S_{i_k} \overset{*}{\Rightarrow} \phi_k$ (that is, $\phi_k \in L(S_{i_k})$), for all $k \in \{1, \ldots, m\}$. Note that, for all k, $depth(w_k) \leq depth(w) - 1$, because we removed the call symbol b_k and its matching return symbol $\overline{b_k}$;
  - $m \in \mathbb{N}$.

Let $w' = u_1 S_{i_1} u_2 S_{i_2} ... S_{i_m} u_{m+1}$, in the language denoted by the regular expression e_j. By the construction of the automaton $\mathcal{A}^{S_j}$ (*see step 3*), this word is in the regular language of the automaton $\mathcal{A}^{S_j}$: $w' \in \widetilde{L}(\mathcal{A}^{S_j})$.

For all k, $\phi_k = b_k w_k \overline{b_k} \in L(S_{i_k})$ and $depth(w_k) \leq depth(w) - 1$. By induction, it holds that $\phi_k \in L(\mathcal{A}^{S_{i_k}})$.

Finally, by Property (2.7), because $u_1 S_{i_1} u_2 ... S_{i_m} u_{m+1} \in \widetilde{L}(\mathcal{A}^{S_j})$ and $\phi_k \in L(\mathcal{A}^{S_{i_k}})$ for all k :

$$aw\overline{a} = a_j u_1 \phi_1 u_2 ... \phi_m u_{m+1} \overline{a_j} \in L(A^{S_j}).$$

- $aw\overline{a} \in L(S_j) \Leftarrow aw\overline{a} \in L(\mathcal{A}^{S_j})$:

  By Definition 2.6, we observe that $aw\overline{a} = f(S_j) w \overline{f(S_j)}$. With *step 4*, we se that $f(S_j) = a_j$. Then, $aw\overline{a} = a_j w \overline{a_j}$.

  To prove the claim, we proceed by induction, using the depth of w (see Definition 2.9).

  Base Case : $depth(w) = 0$**.**

  As $w \in WM(\Sigma_{JSON})$ and $depth(w) = 0$, it holds that w is a word over the internal alphabet Σ_{int}. Since $aw\overline{a} \in L(\mathcal{A}^{S_j})$ and $w \in \Sigma_{int}^*$, w is in the regular language $\widetilde{L}(\mathcal{A}^{S_j})$ (see Property (2.6)).

  The procedural automaton $\mathcal{A}^{S_j}$ is constructed to accept the language denoted by the regular expression e_j (see *step 3*), therefore $\widetilde{L}(\mathcal{A}^{S_j})$ is denoted by the regular expression e_j. As w is in the language denoted by e_j, the derivation $S_j \Rightarrow a_j w \overline{a_j}$ exists and, by Definition 1.7 of CFL, $a_j w \overline{a_j} \in L(S_j)$.

  Induction Step : $depth(w) > 0$**.**

  By the hypothesis $a_j w \overline{a_j} \in L(\mathcal{A}^{S_j})$, it exists a stacked run :

  $$(q_0^{S_j}, \varepsilon) \xrightarrow{w} (q_F^{S_j}, \varepsilon)$$

  where $q_0^{S_j}$ and $q_F^{S_j}$ are respectively the initial state and a final state of $\mathcal{A}^{S_j}$. This stacked run can be decomposed to :

  $$\left(q_0^{S_j}, \varepsilon\right) \xrightarrow{u_1} \left(q_1^{S_j}, \varepsilon\right) \xrightarrow{\phi_1} \left(p_1^{S_j}, \varepsilon\right) \xrightarrow{u_2} \left(q_2^{S_j}, \varepsilon\right) \xrightarrow{\phi_2} ... \xrightarrow{\phi_m} \left(p_m^{S_j}, \varepsilon\right) \xrightarrow{u_{m+1}} \left(q_F^{S_j}, \varepsilon\right),$$

  with :

  - $u_k \in \Sigma_{int}^*$ words over the internal alphabet, for all $k \in \{1, ..., m+1\}$;
  - $\phi_k = b_k w_k \overline{b_k} \in WM(\Sigma_{JSON})$, with $b_k \in \Sigma_{call}$, for all $k \in \{1, ..., m\}$. Note that, for all k, $depth(w_k) \leq depth(w) - 1$, because we removed the call symbol b_k and its matching return symbol ;
  - $q_k^{S_j}, p_k^{S_j}$ states of automaton $\mathcal{A}^{S_j}$, for all $k \in \{1, ..., m\}$;
  - $m \in \mathbb{N}$;

Because $u_k \in \Sigma_{int}$, the stack in the previous run only changes when the VSPA reads a word ϕ_k, and because $\phi_k \in WM(\Sigma_{JSON})$, the stack is the same before and after the VSPA reads ϕ_k. By Definition 2.4, this means that, for all k, the stacked run of $\phi_k = b_k w_k \overline{b_k}$ is of the form :

$$\left(q_k^{S_j}, \varepsilon\right) \xrightarrow{b_k} \left(q_0^{S_{i_k}}, p_k^{S_j}\right) \xrightarrow{w_k} \left(q_F^{S_{i_k}}, p_k^{S_j}\right) \xrightarrow{\overline{b_k}} \left(p_k^{S_j}, \varepsilon\right),$$

with $q_0^{S_{i_k}}$ and $q_F^{S_{i_k}}$ respectively the initial state and a final state of some procedural automaton $\mathcal{A}^{S_{i_k}}$ such that it exists a transition $q_k^{S_j} \xrightarrow{S_{i_k}} p_k^{S_j}$ (see Property (2.8)).

Especially, $f(\mathcal{A}^{S_{i_k}}) = b_k$ and it exists a stacked run of w_k from the initial state of $\mathcal{A}^{S_{i_k}}$ to a final state of $\mathcal{A}^{S_{i_k}}$ that has the same stack at the beginning and the end of the run :

$$\left(q_0^{S_{i_k}}, \varepsilon\right) \xrightarrow{w_k} \left(q_F^{S_{i_k}}, \varepsilon\right).$$

Therefore, for all $k \in \{1, .., m\}$, by Definition 2.6, it holds that $\phi_k = b_k w_k \overline{b_k} \in L(\mathcal{A}^{S_{i_k}})$. Because $depth(w_k) \leq depth(w) - 1$, by induction, $\phi_k \in L(S_{i_k})$.

By Property (2.8), it exists the following run in the Procedural automaton $\mathcal{A}^{S_j}$:

$$q_0^{S_j} \xrightarrow{u_1} q_1^{S_j} \xrightarrow{S_{i_1}} p_1^{S_j} \xrightarrow{u_2} ... \xrightarrow{S_{i_m}} p_m^{S_j} \xrightarrow{u_{m+1}} q_F^{S_j},$$

and therefore, by Definition 1.6 of language of NFAs :

$$u_1 S_{i_1} u_2 ... S_{i_m} u_{m+1} \in \widetilde{L}(\mathcal{A}^{S_j}).$$

In *step 3*, we construct $\mathcal{A}^{S_j}$ such that its regular language $\widetilde{L}(\mathcal{A}^{S_j})$ is denoted by the regular expression e_j. Then, it exists the derivation $S_j \Rightarrow a_j u_1 S_{i_1} u_2 ... S_{i_m} u_{m+1} \overline{a_j}$.

For all k, $\phi_k \in L(S_{i_k})$. Then, by Definition 1.8, it exists a succession of derivation such that $S_{i_k} \overset{*}{\Rightarrow} \phi_k$.

Finally, it exists the derivations $S_j \Rightarrow a_j u_1 S_{i_1} u_2 ... S_{i_m} u_{m+1} \overline{a_j a_j} \overset{*}{\Rightarrow} a_j u_1 \phi_1 u_2 ... \phi_m u_{m+1} \overline{a_j} = aw\overline{a}$. By Definition 1.8, it holds that :

$$aw\overline{a} \in L(S_j).$$

We proved that for all $aw\overline{a} \in WM(\Sigma_{JSON})$, $aw\overline{a} \in L(S_j) \Leftrightarrow aw\overline{a} \in L(\mathcal{A}^{S_j})$. It is now easy to prove that $w \in L(\mathcal{G}) \Leftrightarrow w \in L(\mathcal{A})$. By Definition 1.8 of CFL, the language $L(\mathcal{G})$ is equal to the language of its axiom $L(S_0)$. By definition 2.6 of VSPA languages, the language $L(\mathcal{A})$ is equal to the language of its starting procedural automaton. In *step 4*, we defined the starting symbol of $\mathcal{A}$ as S_0. Therefore, we have $L(\mathcal{A}) = L(\mathcal{A}^{S_0})$.

By the previous claim, we have $L(S_0) = L(\mathcal{A}^{S_0})$. To conclude, it holds that :

$$L(\mathcal{G}) = L(S_0) = L(\mathcal{A}^{S_0}) = L(\mathcal{A}) \Rightarrow L(\mathcal{G}) = L(\mathcal{A}).$$

Chapter 3

Streaming Validation Algorithm of JSON Document using VSPA

In the previous chapter, we saw that we can construct a VSPA that accepts any JSON schema. However, the call rules of VSPAs are nondeterministic. This means that there can be more than one stacked run for the same word, making it not trivial to verify whether the word is accepted or not by the VSPA.

In this chapter, we present a general *streaming validation* algorithm using a VSPA. A validation algorithm for a language is an algorithm that receives a word as input and returns `true` if the word is in the language, and `false` otherwise. In our case, the language is the one accepted by the VSPA. The algorithm is a "streaming" algorithm because it reads the word symbol by symbol, without going back.

In the case of a JSON schema represented as a language, it is impractical to validate a JSON document using a VSPA. The problem is that the key-value pairs inside the objects of the JSON document can be arbitrarily ordered. If we create a VSPA that validates a JSON schema for all possible key orders, it will have exponentially many states.

In this chapter, we present a streaming validation algorithm validating JSON schemas. This algorithm uses a VSPA that validates the JSON schema in a **fixed** order of keys, and a *key graph* that allows validation regardless of the order of key-value pairs.

Finally, we will compare the efficiency of our algorithm with another streaming validation algorithm and with a classical validation algorithm for JSON schema.

3.1 Streaming Validation using a VSPA

In this section, we consider $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S)$ a VSPA (see Definition 2.3) with :

- $\widehat{\Sigma} = \Sigma_i \cup \Sigma_c \cup \Sigma_r \cup \Sigma_p$, the VPSA alphabet (see Definition 2.1) ;
- $\Gamma = \{Q^J | J \in \Sigma_{proc}\}$, the stack alphabet ;
- $\mathcal{P} = \{\mathcal{A}^J | J \in \Sigma_{proc}\}$, the sets of procedural automata (see Definition 2.2) ;
- $f : \Sigma_{proc} \to \Sigma_{call} : f(J) = a$;
- $S \in \Sigma_{proc}$, the start symbol.

3.1.1 Validation Algorithm

This section focuses on a *streaming validation* algorithm for a VSPA. Given a VSPA $\mathcal{A}$ such that $L(\mathcal{A})$ is its language, the validation algorithm receives a word w as input and returns `true` if $w \in L(\mathcal{A})$, `false` otherwise. The algorithm uses the VSPA $\mathcal{A}$ in order to assess if $w \in L(\mathcal{A})$, and is a "streaming" algorithm. By "streaming", we mean that the algorithm reads the symbols of w one by one, and only once.

The idea behind the validation algorithm is similar to the transformation of NFA into DFA (see Appendix A). We start in the initial state with an empty stack. When we read a symbol, we keep track of all possible reachable states and all possible stacks.

We suppose that we want to validate a word $w = a_1 a_2 ... a_n$. We first read the symbol a_1 and assess that $f(S) = a_1$. We start in the initial configuration $\left(q_0^S, \varepsilon\right)$. When we read the symbol a_2, we move into all reachable configurations in the stacked run (see Definition 2.5) $\hat{\delta}(q_0^S, a_2, \varepsilon)$.

Now, we read the symbol a_3. We must move into all reachable configurations from all the previous configurations $\hat{\delta}(q_0^S, a_2, \varepsilon)$. More formally, after reading a_3, we are in the set of configurations $\hat{\delta}(q_0^S, a_2 \cdot a_3, \varepsilon) = \left\{ \hat{\delta}(q, a_3, \gamma) \middle| (q, \gamma) \in \hat{\delta}(q_0^S, a_2, \varepsilon) \right\}$.

We continue like this until we read the symbol a_{n-1}. We then reach the set of configurations $\hat{\delta}(q_0^S, a_2 a_3 ... a_{n-1}, \varepsilon)$. To validate the word, we have to verify that $\overline{f(S)} = a_n$ and that there exists a configuration (p, ε) in the set of reachable configurations, with $p \in F^S$ a final state of the initial procedural automaton $\mathcal{A}^S$.

Because we push (resp. pop) something in the stack word only when we read a call symbol (resp. return symbol), we can store the set of reachable states and the reachable stack words separately. We simplify the "*sets of reachable configurations*" by the configuration $(\mathcal{Q}, \gamma)$, where :

- $\mathcal{Q} \subseteq \left\{Q^J | J \in \Sigma_{proc}\right\}$ is a subset of all states of the VSPA ;
- γ is a stack word, where each of its symbols is a subset of states : $\gamma \in \left\{Q^J | J \in \Sigma_{proc}\right\}^*$.

When we read a symbol, we will follow a generalization of the semantics of VSPAs (see Definition 2.4), but when we read a call symbol, we stack the starting states instead of the arrival states. From a configuration $(\mathcal{Q}, \gamma)$, we define the semantics when we read the symbol $a \in \Sigma$ by :

- *Interior Rules* ($a \in \Sigma_{int}$) :

  Since the interior rule does not modify the stack, it behaves similarly to NFAs. We check, for each state in $\mathcal{Q}$, all the reachable states by reading the symbol a :

  $$(\mathcal{Q}, \gamma) \xrightarrow{a \in \Sigma_{int}} (\mathcal{Q}', \gamma)$$

  with $\mathcal{Q}' = \{q' | \exists q \in \mathcal{Q} : q \xrightarrow{a} q'\}$.

- *Call Rules* ($a \in \Sigma_{call}$) :

  The rule changes a lot compared to what we saw in the semantics of VSPAs (see Definition 2.4). First of all, we push the current set of states on top of the stack (the return handles the transition into the next states of the active procedural automata). Then, for all $J \in \Sigma_{proc}$ such that $f(J) = a$, we go the initial states of $\mathcal{A}^J$:

  $$(\mathcal{Q}, \gamma) \xrightarrow{a \in \Sigma_{call}} (\mathcal{Q}', \mathcal{Q} \cdot \gamma)$$

  with $\mathcal{Q}' = \left\{q_0^J | \exists J \in \Sigma_{proc} : f(J) = a\right\}$.

- *Return Rules* ($a \in \Sigma_{return}$) :

  When we read a return symbol, the top of the stack contains the set states $\mathcal{Q}''$, where the VSPA was when it reads the last unmatched call symbol. From this call symbol, it has read a word w. If this w made an accepting stacked run in a procedural automaton $\mathcal{A}^J$, we should follow all the transitions $\mathcal{Q}'' \xrightarrow{J} \mathcal{Q}'$, if the procedural symbol corresponds to the return symbol : $\overline{f(J)} = a$.

  The return rule first checks, for each symbol $J \in \Sigma_{proc}$, if $\overline{f(J)} = a$ and if the current set of states contains a final state of $\mathcal{A}^J$. If it does, it performs, from all states on the top of the stack, the transitions reading the symbol J :

  $$(\mathcal{Q}, \mathcal{Q}'' \cdot \gamma) \xrightarrow{a \in \Sigma_{ret}} (\mathcal{Q}', \gamma)$$

  with $\mathcal{Q}' = \left\{q' | \exists J \in \Sigma_{proc} : a = \overline{f(J)}, \mathcal{Q} \cap F^J \neq \emptyset, \exists q'' \in \mathcal{Q}'' : q'' \xrightarrow{J} q'\right\}$.

We can now construct our algorithm describing the validation of a word with a JSPA $\mathcal{A}$ based on these three rules (see Algorithm 3.1).

The first lines of the algorithm describe the initialization. They verify that the first symbol of the word corresponds to the call symbol of the starting procedural automaton and initialize $\mathcal{Q}$ to $\{q_0^S\}$ and γ to ε.

Then, it loops over each symbol of w. It creates $\mathcal{Q}'$, which will be the updated set of reachable states after reading the symbol.

When it reads an internal symbol a_{int}, for all symbols $q^J \in \mathcal{Q}$, it adds to $\mathcal{Q}'$ all the states p^J where it exists a transition $q^J \xrightarrow{a_{int}} p^J$.

When it reads a call symbol a_{call}, it first pushes $\mathcal{Q}$ to the stack. For all procedural symbols in Σ_{proc}, it checks if a_{call} is the call symbol of the procedural symbol. If it does, it adds to $\mathcal{Q}'$ the initial state of the procedural automaton denoted by the procedural symbol.

When it reads a return symbol a_{return}, it first pops $\mathcal{Q}''$ from the stack. For all procedural symbol J, it checks if $\overline{f(J)}$ corresponds to the return symbol a_{return}, and if there is a final state of $\mathcal{A}^J$ in $\mathcal{Q}$. If it does, it adds to $\mathcal{Q}'$ all the reachable states reading J from the states of $\mathcal{Q}''$.

Finally, the algorithm checks if the last symbol of the word corresponds to the return symbol of the starting procedural automaton, if $\mathcal{Q}$ contains a final state of the starting procedural automaton and if the stack is empty. If it does, the word is accepted.

Note that we can slightly improve the algorithm when we read a call symbol. If, in the current set of states $\mathcal{Q}$, there exists no transition reading a procedural symbol $J \in \Sigma_{proc}$, it is not necessary to put in the next set of states $\mathcal{Q}'$ the initial state of the procedural automaton $\mathcal{A}^J$, even if $f(J) = a$.

Algorithm 3.1 Streaming Validation of a Word using a VSPA

Require : $\mathcal{A} = (\Sigma, \Gamma, \mathcal{P}, f, S)$ a VSPA and a word $w = a_1...a_n \in \Sigma^*$.
Ensure : `true` is returned if $w \in L(\mathcal{A})$, `false` instead.
$\mathcal{Q} \leftarrow \{q_0^S\}$
$\gamma \leftarrow \varepsilon$
for $i = 1 : n$ **do**
    $\mathcal{Q}' \leftarrow \{\}$
    if $a_i \in \Sigma_{int}$ **then**
        for $q \in \mathcal{Q}$ **do**
            $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \delta(q, a_i)$
        end for
    else if $a_i \in \Sigma_{call}$ **then**
        $\gamma.push(\mathcal{Q})$
        for $K \in \Sigma_{proc}$ **do**
            if $f(K) = a_i$ **then**
                $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{q_0^K\}$
            end if
        end for
    else if $a \in \Sigma_r$ **then**
        $\mathcal{Q}'' = \gamma.pop()$
        for $J \in \Sigma_{proc}$ **do**
            if $\overline{f(J)} = a_i$ **and** $F^J \cap \mathcal{Q} \neq \emptyset$ **then**
                for $q'' \in \mathcal{Q}''$ **do**
                    $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \delta(q'', J)$
                end for
            end if
        end for
    end if
    $\mathcal{Q} \leftarrow \mathcal{Q}'$
end for
return $\mathcal{Q} \cap F^S \neq \emptyset$ **and** $\gamma = \varepsilon$

3.1.2 Example of Validation with JSPA

As an example, we take the VSPA described in Figure 2.4. Here are some words and how they are validated by Algorithm 3.1 :

- $w_0 = az$:

  This one is trivial, as it's length is 2. We first check that the first symbol $a = f(S)$ and we start in the configuration $(\mathcal{Q}, \gamma) = (\{q_0^S\}, \varepsilon)$. Then we do not iterate in the loop (because $|w_0| = 2$), and we check if the last symbol $z = \overline{f(S)}$, if $\mathcal{Q}$ contains a final state of $\mathcal{A}^S$, and if the stack γ is empty. As all these conditions are respected, the word is validated.

- $w_1 = aabzz$:

  Like the previous word, we check the first symbol and start in configuration $(\mathcal{Q}, \gamma) = (\{q_0^S\}, \varepsilon)$.

  We then read the call symbol a. We push $\mathcal{Q} = \{q_0^S\}$ on top of the stack, and we go to the set of all initial states of procedural automata that reads a as a call symbol, in this case $\mathcal{A}^S$ and $\mathcal{A}^R$. After reading this symbol, we are in the configuration $\left(\{q_0^S, q_0^R\}, \{q_0^S\}\right)$.

  Now, we have to read the internal symbol b. We check all transitions reading b from the states in $\mathcal{Q} = \{q_0^S, q_0^R\}$. From the state q_0^S, there is no transition reading b. From the state q_0^R, it exists a transition reading b that goes to the state q_1^R. After reading the symbol b, we are in the configuration $\left(\{q_1^R\}, \{q_0^S\}\right)$.

  The next symbol to read is the return symbol z. We actually are in the state q_1^R, which is a final state of $\mathcal{A}^R$. We pop $\{q_0^S\}$ from the stack and we check all transitions from the state q_0^S reading R (because we were in a final state of $\mathcal{A}^R$). As it exists the transition $q_0^S \xrightarrow{R} q_0^S$, we go to the configuration $(\{q_0^S\}, \varepsilon)$.

  Finally, we check if the last symbol $z = \overline{f(S)}$, if we are in a final state of $\mathcal{A}^S$, and if γ is empty. As all these conditions are respected, the word is validated.

- $w_2 = aaza$:

  The first two symbols of this word are the same as w_1. After reading them, we are in the configuration $\big(\{q_0^S, q_0^R\}, \{q_0^S\}\big)$.

  We then read the return symbol z. We are in the set of two states $\{q_0^S, q_0^R\}$, but only q_0^S is a final state of its procedural automaton. When we pop the set $\{q_0^S\}$ from the stack, we only look at the transitions reading the symbol S. As it exists the transition $q_0^S \xrightarrow{S} q_0^S$, we go to the configuration $(\{q_0^S\}, \varepsilon)$.

  Finally, we read the last symbol z, and we have already seen that the word is accepted in the configuration $(\{q_0^S\}, \varepsilon)$.

- $w_3 = aaz$:

  The first two symbols are the same as w_1. After reading them, we are in the configuration $\big(\{q_0^S, q_0^R\}, \{q_0^S\}\big)$.

  We now have to read the last symbol z. A final state of $\mathcal{A}^S$ is in the set $\{q_0^S, q_0^R\}$, but the stack is not empty. Therefore, the word is rejected.

- $w_5 = azz$:

  We read the first symbol and start in the configuration $(\{q_0^S\}, \varepsilon)$.

  We then read a return symbol z, but the stack is empty. Then, we can already reject this word.

  We could suppose that we can pop an empty set from the stack, and go to the configuration $(\emptyset, \varepsilon)$. Anyway, we would reject the word as it does not contain a final state of $\mathcal{A}^S$ when we read its last symbol.

- $w_4 = abz$:

  We read the first symbol and start in the configuration $(\{q_0^S\}, \varepsilon)$.

  Then, we read the internal symbol b. As there exists no transition reading b from q_0^S ($\delta(q_0^S, b) = \emptyset$), we can already reject the word. To be more formal, we go to the configuration $(\{\}, \varepsilon)$. This configuration does not contain a final state of $\mathcal{A}^S$ in its set of states, and is therefore rejected when we read the last symbol.

- $w_5 = za$:

  As w_5 does not begin with the symbol $f(S) = a$, w_5 is immediately rejected.

With these examples, we can see that the word is rejected in four cases. There are actually six cases, but the last two cases couldn't be illustrated with the VSPA described in Figure 2.4 :

- At some point of the reading, we are in the configuration $(\{\}, \gamma)$, where Ω is empty ;
- We read a return symbol when the stack is empty. This means that the word is not Well-Matched (see Definition 2.7) ;
- The stack is not empty when we read the last symbol ;
- The first symbol is not $f(S)$;

- We are not in a final state of $\mathcal{A}^S$ when we read the last symbol ;
- The last symbol is not $\overline{f(S)}$.

Any rejected word will, at some point in its reading, reach (at least) one of these cases.

Some of these cases are not handled by Algorithm 3.1. We can add *early-stopping* in the algorithm to improve its computation time for invalid document :

- If, at the end of the for-loop of the algorithm, the set of states $\mathcal{Q}$ is empty, we can reject the word.
- If, when reading a return symbol, the stack is empty, we can reject the word.

3.1.3 Closed Schema or Fixed Order of Keys

We could apply this algorithm directly for JSON schemas. However, there is a big flow with the procedural automata accepting object schema.

As seen in Section 1.4.3, a CFG must be closed over object schemas. This means that, whenever it contains a productions $S \to \langle k_1 S_1 \# ... \# k_m S_m \rangle$, it also contains all production $S \to \langle k_{i_1} S_{i_1} \# ... \# k_{i_m} S_{i_m} \rangle$, where $(i_1, ..., i_m)$ is any permutation of $(1, ..., m)$.

A closed CFG may have a lot of productions only to describe an object. In fact, when an object has m keys, a CFG will need $m!$ productions to describe it. When the productions are transformed to a unique regular expression with Theorem 1.3, the regular expression needs to contain the disjunction of all $m!$ permutations of $(1, ..., m)$. Creating a finite automaton accepting this regular expression results in an automaton with $\mathcal{O}(m!)$ states.

Having so many states is usually impractical. One way to reduce the number of states is to assume that the CFG is not closed over object schemas. This means that each object schema has a fixed key order, and a JSON document must follow this order to be considered valid. Suppose that a JSON document satisfies the schema but uses a different key order than the one defined in the CFG. In that case, it will not belong to the language of the CFG. If we construct a VSPA accepting this unclosed CFG, the VSPA may reject documents that do satisfy the original schema, simply because they do not follow the fixed key order imposed by the CFG.

To use Algorithm 3.1 in the application of JSON schemas, we consider one of the following assumptions:

- The CFG is closed over object schemas, and the procedural automata describing object schemas have $\mathcal{O}(m!)$ states.
- The CFG is not closed over object schemas, and the JSON documents must have a fixed order of keys in order to be validated.

In Section 3.2, we propose an algorithm that uses a VSPA accepting the language of a CFG **unclosed** over object schema, but which can still validate JSON documents with **any key order**.

3.2 VSPA Validation for Closed Schema

In this section, we consider :

- $\Sigma_{JSON} = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$ a JSON alphabet, with $\Sigma_{int} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#\}$, $\Sigma_{call} = \{\langle, \sqsubset\}$ and $\Sigma_{ret} = \{\rangle, \sqsupset\}$;
- $\mathcal{G} = \{\mathcal{S}, \Sigma_{JSON}, P, S_0\}$ a **closed** extended CFG describing a JSON schema, of the form described by Theorem 1.3 :
  - The left-hand sides of productions in P are pairwise distinct ;
  - For all $S_i \in \mathcal{S}$, its production is of the form $S_i \to a_i e_i \overline{a_i}$, with $a_i \in \Sigma_{call}$ and e_i a regular expression over $\widetilde{\Sigma} = \Sigma_{int} \cup \mathcal{S}$;

As seen before, a CFG must be closed over object schemas, meaning that whenever it contains a productions $S \to \langle k_1 S_1 \# ... \# k_2 S_2 \rangle$, it contains all productions for all permutation of keys. This results in a VSPA with exponentially many states, which is unpractical.

In this section, we propose an algorithm that validates a JSON document using a VSPA without this problem. The approach we use is inspired by an approach already used to validate JSON documents using a *Visibly Pushdown Automaton* [8]. The algorithm uses a *key graph* to allow arbitrary orders of keys in the JSON document.

Based on Theorem 2.1, we build $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \mathcal{P}, f, S_0)$ a VSPA (see Definition 2.3) with :

- $\widehat{\Sigma} = \Sigma_{JSON} \cup \mathcal{S}$ the VPSA alphabet (see Definition 2.1), with $\mathcal{S} = \Sigma_{proc}$;
- $\Gamma = \left\{Q^{S_j} | S_j \in \mathcal{S}\right\}$ the stack alphabet ;
- $\mathcal{P} = \left\{\mathcal{A}^{S_j} | S_j \in \mathcal{S}\right\}$ the sets of all procedural automata .
- $f : \mathcal{S} \to \Sigma_{call} : f(S_i) = a_i$;
- $S_0 \in \mathcal{S}$ the start symbol.

such that $L(\mathcal{A}) \subseteq L(\mathcal{G})$ accepts all JSON documents that satisfy the JSON schema for an arbitrary fixed order of keys.

3.2.1 Key Graph

This section introduces the *key graph*, that helps us to allow any key order with the VSPA $\mathcal{A}$. Each procedural automaton describing an object schema has its own key graph.

To define the vertices of the graph of a procedural automaton, we look at every valid key-value pair $k \cdot v$. For all pairs, we create a vertex of the form (p, k, p') when it exists the run $p \xrightarrow{k \cdot v} p'$ in the procedural automaton. We add an edge between two vertices (p_1, k_1, p_1') and (p_2, k_2, p_2') when it exists the transition $p_1' \xrightarrow{\#} p_2$. Since $\mathcal{A}^{S_j}$ only describes an object schema, the key graph corresponding to $\mathcal{A}^{S_j}$ usually has a form similar to the automaton.

Definition 3.1: Key Graph

The Key Graph of a procedural automaton $\mathcal{A}^{S_j}$, noted G^{S_j} has :

- The vertices (p,k,p'), with $p,p' \in Q^{S_j}$ and $k \in \Sigma_{key}$, if there exist in $\mathcal{A}^{S_j}$ the transitions :

$$p \xrightarrow{k} q \xrightarrow{v} p',$$

  with $v \in \Sigma_{pVal} \cup \mathcal{S}$ and $q \in Q^{S_j}$;

- the edges $((p_1,k_1,p'_1),(p_2,k_2,p'_2))$ if there exists the transition $p'_1 \xrightarrow{\#} p_2$ in $\mathcal{A}^{S_j}$.

We can define the following property of the key graph :

Let G^{S_j} the key graph of $\mathcal{A}^{S_j}$. There exists a path $((p_1,k_1,p'_1),(p_2,k_2,p'_2),...,(p_n,k_n,p'_n))$ in G^{S_j} with $p_1 = q_0^{S_j}$ the initial state of $\mathcal{A}^{S_j}$ and p'_n a final state of $\mathcal{A}^{S_j}$ if and only if :

$$k_1v_1\#k_2v_2\#...\#k_nv_n \in \widetilde{L}(\mathcal{A}^{S_j}),$$

with $v_i \in \Sigma_{int} \cup \mathcal{S}$ such that it exists the path $p_i \xrightarrow{k_iv_i} p'_i$ in $\mathcal{A}^{S_j}$. If such a path exists, all keys k_i with $i \in \{1,...,n\}$ are pairwise distinct.

This property is easy to prove. As it exists the path $((p_1,k_1,p'_1),(p_2,k_2,p'_2),...,(p_n,k_n,p'_n))$, by Definition 3.1 of key graph, it exists the transitions $p_i \xrightarrow{k_iv_i} p'_i$ (for all $i = 1,...,n$) and $p'_i \xrightarrow{\#} p_{i+1}$ (for all $i = 1,...,n-1$) in $\mathcal{A}^{S_j}$. As p_1 is the initial state and p'_n a final state of $\mathcal{A}^{S_j}$, it exists an accepting run of $k_1v_1\#...\#k_nv_n$.

Note that, in such a path, all the keys are pairwise distinct, because there exists no accepting run of a word $w \in \widetilde{\Sigma}$ with two same keys. If it would, the language of the VSPA wouldn't correspond to the JSON schema.

As an example of construction of key graph, we take the following production representing an unclosed object schema :

$$S \to \langle (k_1\texttt{i}\#k_2\texttt{i}) + (k_1\texttt{s}\#k_2\texttt{s}) \rangle.$$

The corresponding procedural automaton $\mathcal{A}^S$ accepting this schema is given in Figure 3.1.

It is easy to construct the key graph for this procedural automaton. The possible value paired with k_1 are `i` and `s`. The possible run for k_1`i` is $q_0^S \xrightarrow{k_1} q_1^S \xrightarrow{\texttt{i}} q_2^S$, so we add the vertex (q_0^S,k_1,q_2^S). For the key-value k_1`s`, the possible run is $q_0^S \xrightarrow{k_1} q_1^S \xrightarrow{\texttt{s}} q_5^S$ and we add the vertex (q_0^S,k_1,q_5^S). Likewise with k_2, we add the vertices (q_3^S,k_2,q_8^S) and (q_6^S,k_2,q_8^S). Finally, as there are the transitions $q_2^S \xrightarrow{\#} q_3^S$ and $q_5^S \xrightarrow{\#} q_6^S$, we add the edges between vertices (q_0^S,k_1,q_2^S) and (q_3^S,k_2,q_8^S), and between (q_0^S,k_1,q_5^S) and (q_6^S,k_2,q_8^S). The resulting key-graph can be seen in Figure 3.2.

We will see later how we use this key graph to validate an object schema in any order of keys.

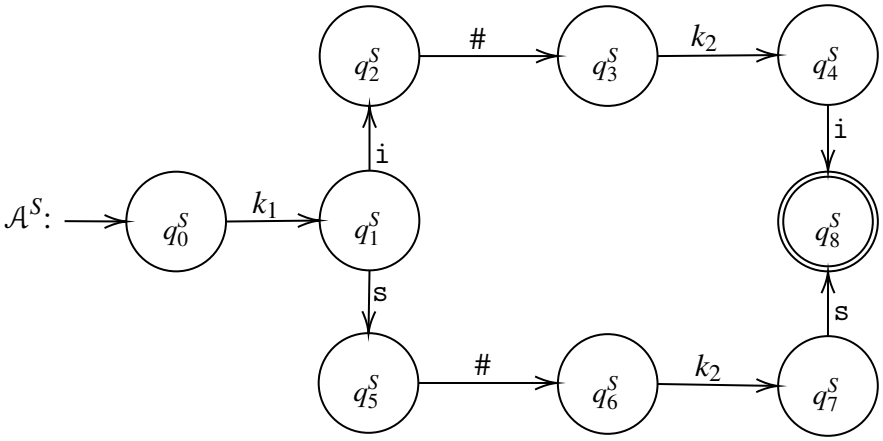


Figure 3.1: Example of Procedural Automaton.

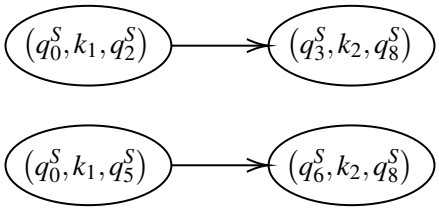


Figure 3.2: Key Graph corresponding to Procedural Automaton in Figure 3.1.

Computation of the Key Graph

We provide here a simple polynomial time algorithm to generate a key graph G^S of a procedural automaton $\mathcal{A}^S = (Q^S, q_0^S, \widetilde{\Sigma}, F^S, \delta^S)$, with $\widetilde{\Sigma} = \Sigma_{int} \cup \mathcal{S}$. We suppose that $\mathcal{A}^S$ is nondeterministic.

We take an arbitrary state $p \in Q^S$ and an arbitrary key $k \in \Sigma_{key}$. For all symbols $v \in \Sigma_{pVal} \cup \mathcal{S}$, we can add to the key graph the vertices (p, k, p'), with $p' \in \widehat{\delta}^S(p, k \cdot v) = \delta^S(\delta^S(p,k), v)$. We do this for all states $p \in Q^S$ and for all keys $k \in \Sigma_{key}$ to obtain the sets of vertices of the key graph.

Finally, for all pairs of vertices (p_1, k_1, p_1') and (p_2, k_2, p_2'), if $p_2 \in \delta(p_1', \#)$, we add the edge between these vertices.

This method is described in Algorithm 3.2. It is trivial to see that the time complexity of this algorithm is :

$$\mathcal{O}\left(|Q^S|^3 \times |\Sigma_{key}| \times |\Sigma_{pVal} \cup \mathcal{S}| + |Q^S|^4 \times |\Sigma_{key}|^2\right). \tag{3.1}$$

The number of vertices is $\mathcal{O}\left(|Q^S|^2 \times |\Sigma_{key}|\right)$, which are computed in $\mathcal{O}\left(|Q^S|^3 \times |\Sigma_{key}| \times |\Sigma_{pVal} \cup \mathcal{S}|\right)$, and the number of edges is $\mathcal{O}(|V|^2)$, where $|V|$ is the number of vertices of G^S.

Note that, in practice, a procedural automaton describing an object schema typically has only a few paths from its initial state to a final state. Therefore, the key graph for such a procedural automaton has approximately $\mathcal{O}(|Q^S|)$ vertices and edges. In this case, it is useful to implement a more efficient algorithm to compute the key graph. A simple approach is a variant of graph exploration starting from the initial state of the procedural automaton, which creates a vertex for a key whenever it finds a path that reads a key-value pair. Note that the theoretical time complexity of such an algorithm remains the same as the time complexity of Algorithm 3.2.

Algorithm 3.2 Computation of a Key Graph for a Procedural Automaton of a VSPA

Require : $A^S = (Q^S, q_0^S, \widetilde{\Sigma}, F^S, \delta^S)$ a procedural automaton, with $\widetilde{\Sigma} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#\} \cup \mathcal{S}$.
Ensure : $G^S = (V, E)$ the corresponding key graph of $\mathcal{A}^S$, where V is the set of vertices and E the set of edges.
$V \leftarrow \{\}$
for $p \in Q^S$ **do**
  for $k_i \in \Sigma_{key}$ **do**
    for $v \in \Sigma_{pVal} \cup \mathcal{S}$ **do**
      for $q \in \delta^S(p, k_i)$ **do**
        for $p' \in \delta^S(q, v)$ **do**
          $V \leftarrow V \cup (p, k_i, p')$
        end for
      end for
    end for
  end for
end for
$E \leftarrow \{\}$
for $(p_1, k_i, p_1') \in V$ **do**
  for $(p_2, k_j, p_2') \in V$ **do**
    if $p_2 \in \delta(p_1', \#)$ **then**
      $E \leftarrow E \cup ((p_1, k_i, p_1'), (p_2, k_j, p_2')$
    end if
  end for
end for
$G^S \leftarrow (V, E)$

3.2.2 Streaming Algorithm

We propose a validation algorithm of a word $w \in L(\mathcal{G})$, with $\mathcal{G}$ a CFG describing a JSON schema. The algorithm uses a VSPA $\mathcal{A}$ that accepts all JSON documents that satisfy the JSON schema for a fixed order of keys, $L(\mathcal{A}) \subseteq L(\mathcal{G})$. The algorithm uses the key graph and is inspired by an algorithm used for VPA [8].

During the reading of an object, we store some variables that keep track which keys has been read, and which vertices of the key graph are valid :

- $R \subseteq \{Q^{S_j} \times Q^{S_j} | S_j \in \mathcal{S}\}$, a subset of pairs of states. Each pair of states represents the start and end states of the reading of a key-value pair ;
- $K \subseteq \Sigma_{key}$, a set containing all keys already seen in the object ;
- $k_n \in \Sigma_{key}$, the current key that we are reading ;
- $Good = \{Good_{S_j} | S_j \in \mathcal{S}\}$, the vertices of the key graph G^{S_j} that are marked as "possible".

Let us explain how these variables work. Suppose we have already read the word $z \cdot a \cdot u \in \Sigma_{JSON}^*$, with $u \in WM(\Sigma_{JSON})$ and $a \in \Sigma_{call}$ the last unmatched call symbol read :

- If $a = \sqsubset$, then all these values are set to *null*.
- If $a = \lessdot$, then $u = k_1v_1\#...\#k_{n-1}v_{n-1}\#u'$ with $k_i \in \Sigma_{key}$, $v_i \in WM(\Sigma_{JSON})$ and u' is a prefix of k_nv_n, which represent the key-value pair we are reading. Then :
  - $R \subseteq \left\{ \left(q^{S_j}, p^{S_j}\right) | \exists \left(q^{S_j}, \gamma\right) \xrightarrow{u'} \left(p^{S_j}, \gamma\right) \right\}$, a subset of all possible stacked runs of u'. Note that R only focuses on the key-value pair k_nv_n we are reading.
  - $K = \{k_1, ..., k_{n-1}, k_n\}$ contains all the keys read in u (note that, if $u' = \varepsilon$, K contains the key next to u) ;
  - k_n is the current key that's read ;
  - $Good \subseteq \{(p,k,p') | \exists S_j \in \mathcal{S}, (p,k,p') \in G^{S_j}\}$, is a subset of the vertices of all the key graphs. For a key-value pair k_iv_i in u (with $i \in \{1, ..., n-1\}$), the vertex (p_i, k_i, p'_i) is in *Good* if the stacked run $(p_i, \gamma) \xrightarrow{k_iv_i} (p'_i, \gamma)$ exists in the VSPA.

Not to lose these values whenever we read a call symbol, we store these values in the stack, alongside the set of state $\mathcal{Q}$ as defined in Section 3.1.1.

After reading a whole object $\lessdot k_1v_1\#...\#k_nv_n\gtrdot$, we have to use these variables to see if the word is valid. We use the set K, which contains all the keys read, and the set *Good*, which contains all the vertices of the key graphs which corresponds to a possible stacked run of a key-value pair k_iv_i. We create the function *Valid* that take K and *Good* as input and return a set of procedural symbols. This set contains a procedural symbol $S_j \in \mathcal{S}$ if it exists a valid path in the key-graph G^{S_j}. A path in G^{S_j} is valid if all its vertices are in *Good*, all the keys in K appear exactly once in the path, and the path starts in an initial state and ends in a final state.

Definition 3.2: Valid Function

We define the function $Valid(K, Good) \subseteq \mathcal{S}$ as the set of all procedural symbols whose corresponding key graph contains a valid path, with $K \subseteq \Sigma_{key}$ is a set of keys and *Good* is a set of vertices.
A path $((p_1,k_1,q_1), ..., (p_n,k_n,q_n))$ in the key graph G^{S_j} is considered valid if and only if :

- p_1 and q_n are respectively the initial and a final state of $\mathcal{A}^{S_j}$;
- All vertices are in *Good* : $\forall i = 1, ..., n : (p_i, k_i, q_i) \in Good$;
- The path contains all keys in K : $\{k_1, ..., k_n\} = K$.

Let us explain the valid function. Suppose that there is a valid path $((p_1,k_1,q_1), ..., (p_n,k_n,q_n))$ in the key graph G^{S_j}. As all vertices (p_i, k_i, q_i) are in Good, this means that there the stacked run $(p_i, \gamma) \xrightarrow{k_iv_i} (q_i, \gamma)$ was performed by the VSPA, for some value $v_i \in WM(\Sigma_{JSON})$. Because there exists the valid path, this means that it exists the stacked run :

$$(p_1, \gamma) \xrightarrow{k_1v_1} (q_1, \gamma) \xrightarrow{\#} (p_2, \gamma) \xrightarrow{k_2v_2} (q_2, \gamma) \xrightarrow{\#} ... \xrightarrow{\#} (p_n, \gamma) \xrightarrow{k_nv_n} (q_n, \gamma),$$

with p_1 and q_n are respectively the initial and a final state of $\mathcal{A}^{S_j}$. Since there is no stack change, the

word $k_1v_1\#...\#k_nv_n$ is accepted, $k_1v_1\#...\#k_nv_n \in L(\mathcal{A}^{S_j})$. The valid function doesn't look at the order of the keys. Therefore, the word is considered accepted, whatever the permutation of key-value pairs.

Note that the implementation of the function *Valid* uses a slightly modified Depth-First Search (DFS) algorithm [9] to find if a valid path exists in a key graph. DFS is a graph exploration algorithm. This work will not further elaborate on the practical implementation of this function.

Before formally explaining how the algorithm works, let us first give a small illustrative example. Consider the procedural automaton $\mathcal{A}^S$ (see Figure 3.1), its key graph G^S (see Figure 3.2) and suppose that we want to validate the word $w = \langle k_2\mathtt{s}\#k_1\mathtt{s}\rangle$.

When reading the first call symbol $\langle$, we cannot start from the configuration (q_0^S, ε), because there is no transition reading k_2 from that state. Instead, we look at each vertex of the key graph G^S that can read k_2, in this case (q_3^S, k_2, q_8^S) and (q_6^S, k_2, q_8^S). We therefore move to the set of states $\mathcal{Q} = \{q_3^S, q_6^S\}$, that is, all the states that can read k_2.

When reading the symbols $k_2 \cdot \mathtt{s}$, we first move to the set of states $\{q_4^S, q_7^S\}$, and then to the set of states $\{q_8^S\}$. This is where the variable R comes in. We need to know that the path we used is $q_6^S \xrightarrow{k_2} q_7^S \xrightarrow{\mathtt{s}} q_8^S$. R is update after each symbol to remember the possible paths.

We then need to read the symbol #. This symbol is special in our algorithm, as it does not perform any transition in the automaton. Instead, we first register the valid path taken for the previous case. In this case, we store in *Good* the vertex (q_6^S, k_2, q_8^S), as it represents the path we took. Next, we look at the next symbol in the word, that should be a key. In our case, the symbol is k_1. As we did before with k_2, we look at each vertex of the key graph corresponding to k_1, and we move in the set of all the states that can read it, here $\mathcal{Q} = \{q_0^S\}$.

We then perform all the transition reading $k_1 \cdot \mathtt{s}$, to move in the set of states $\{q_5^S\}$ via the path $q_0^S \xrightarrow{k_1} q_1^S \xrightarrow{\mathtt{s}} q_5^S$.

Finally, we read the symbol $\rangle$. As before, we update the variable *Good*, which becomes the set $\{(q_6^S, k_2, q_8^S), (q_0^S, k_1, q_5^S)\}$. To finally validate that the word is accepted by the procedural automaton $\mathcal{A}^S$, we check whether there exists a valid path in the key graph that uses the vertices in *Good*, that is, if $Valid(K, Good)$ (with $K = \{k_2, k_1\}$ the set of keys read) returns the symbol S. In our case, we have the following valid path, which means that the word is accepted :

$$(q_0^S, k_1, q_5^S) \rightarrow (q_6^S, k_2, q_8^S).$$

We can now explain how the algorithm reads each symbol. We talk later about the initialization of the variables. We suppose the VSPA already is in a configuration $(\mathcal{Q}, \gamma)$, where $\mathcal{Q}$ is a set of states, and γ is the stack word. The variables are R, K, k and *Good*. The VSPA now reads the symbol a. The rules to process this symbol are :

- **Case 1 ($a = \sqsubset$) :**

  This means that we process a new array. We go to the set of initial states of arrays procedural automata, and we push $(\mathcal{Q}, R, K, k, Good)$ on top of the stack :

$$(\mathcal{Q}, \gamma) \xrightarrow{a=\sqsubset} (\mathcal{Q}_{upd}, (\mathcal{Q}, R, K, k, Good) \cdot \gamma),$$

with $\mathcal{Q}_{upd} = \left\{ q_0^{S_j} | f(S_j) = \sqsubset \right\}$ the set of all initial states of procedural automata with the corresponding call symbol $\sqsubset$.

The variables R, K, k, $Good$ are all updated to *null*.

- **Case 2 ($a = \langle$) :**

  This means that we process a new object. First, we push $(\mathcal{Q}, R, K, k, Good)$ on the stack. Then, we look ahead to the next symbol b of the word.

  - In the special case where $b = \rangle$, we are reading an empty object $\langle\rangle$. In this case, we process like an array in case 1. We push $(\mathcal{Q}, R, K, k, Good)$ on the stack, we go to the set of states $\mathcal{Q}_{upd} = \{q_0^{S_j} | f(S_j) = \langle\}$, and we update the variables R, K, k, $Good$ to *null*.
  - In the general case where $b \in \Sigma_{key}$, we begin to read a non-empty object. We must look in all the key graphs G^{S_j} for the vertices (p, b, q) and we go to the configuration :

    $$(\mathcal{Q}, \gamma) \xrightarrow{a=\langle} (\mathcal{Q}_{upd}, (\mathcal{Q}, R, K, k, Good) \cdot \gamma),$$

    where $\mathcal{Q}_{upd} = \{p | \exists (p, b, q) \in G^{S_j}, S_j \in \mathcal{S}\}$.
    Finally, we update the value of R, K, k and $Good$ respectively to $R_{upd} = \{(p, p) | p \in \mathcal{Q}_{upd}\}$, $K_{upd} = \{b\}$, $k_{upd} = b$ and $Good_{upd} = \{\}$.
  - In any other cases ($b \notin \Sigma_{key} \cup \{\rangle\}$), the word is rejected.

- **Case 3 ($a = \Sigma_{int} \setminus \{\#\}$) :**

  We process an internal symbol in an array or an object. We follow the transitions of all the states in $\mathcal{Q}$:

  $$(\mathcal{Q}, \gamma) \xrightarrow{a} (\mathcal{Q}_{upd}, \gamma),$$

  with $\mathcal{Q}_{upd} = \{q_{upd} | q \in \mathcal{Q}, q \xrightarrow{a} q_{upd}\}$.

  Additionally, we update the variable R to $R_{upd} = \{(p, q_{upd}) | \exists (p, q) \in R, q \xrightarrow{a} q_{upd}\}$ *(note that, in the case of arrays, R is null and does not need to be updated)* .

- **Case 4 ($a = \#$) :**

  Two possible cases are possible.

  - In the first case, if $R = K = k = Good = null$, this means that we are processing an array. We process as in case 3 :

    $$(\mathcal{Q}, \gamma) \xrightarrow{\#} (\mathcal{Q}_{upd}, \gamma),$$

    with $\mathcal{Q}_{upd} = \{q_{upd} | q \in \mathcal{Q}, q \xrightarrow{\#} q_{upd}\}$ (*as $R = null$, it does not need to be updated*).
  - In the second case, this means that we are processing an object. We just finished reading a key-value pair $k_i v_i$, and we will start to read a new key-value pair of the object.

    The variable R is storing all the possible stacked runs of $k_i v_i$: $\forall (p, q) \in R, \exists (p, \varepsilon) \xrightarrow{k_i v_i} (q, \varepsilon)$. We update the variable $Good$ to $Good_{upd} = Good \cup \{(p, k_i, q) | (p, q) \in R\}$. This means that all the vertex (p, k_i, q) represents a possible path in the VSPA for the word we are reading.

Then, we look ahead to the next symbol b of the word. If $b \notin \Sigma_{key}$ or if $b \in K$ – meaning we have a repetition of keys – the word is rejected.

We process similarly as in case 2. We look in the key graphs for the vertices (p,b,q) and we go to the configuration :

$$(\mathcal{Q},\gamma) \xrightarrow{\#} (\mathcal{Q}_{upd},\gamma),$$

where $\mathcal{Q}_{upd} = \{p | \exists (p,b,q) \in G^{S_j}, j \in \{0,...,m\}\}$.

Finally, we update the value of R, K and k respectively to :

* $R_{upd} = \{(p,p) | p \in \mathcal{Q}\}$;
* $K_{upd} = K \cup \{b\}$ and $k_{upd} = b$ (*Good* has already been updated earlier).

- **Case 5 ($a = \sqsupset$) :**

This means that we finished reading an array. The top of the stack should be composed of $(\mathcal{Q}_{stk}, R_{stk}, K_{stk}, k_{stk}, Good_{stk})$. Like we did in the general validation algorithm of VSPA (see Section 3.1.1), we check all the final states in $\mathcal{Q}$ and we process the transitions from $\mathcal{Q}_{stk}$ that reads the procedural symbol of the final states. We pop the top of the stack and go to the configuration :

$$(\mathcal{Q}, (\mathcal{Q}_{stk}, R_{stk}, K_{stk}, k_{stk}, Good_{stk}) \cdot \gamma) \xrightarrow{\sqsupset} (\mathcal{Q}_{upd},\gamma),$$

where $\mathcal{Q}_{upd} = \left\{ q_{upd} | \exists q \in \mathcal{Q}, q \in F^{S_j}, f(S_j) = \sqsubset, \exists q_{stk} \in \mathcal{Q}_{stk}, q_{stk} \xrightarrow{S_j} q_{upd} \right\}$.

Finally, we update R, K, k and *Good* respectively to :

– $R_{upd} = \left\{ (p_{stk}, q_{upd}) | \exists (p_{stk}, q_{stk}) \in R_{stk}, \exists q \in \mathcal{Q}, q \in F^{S_j}, f(S_j) = \sqsubset, q_{stk} \xrightarrow{S_j} q_{upd} \right\}$;

– $K_{upd} = K_{stk}$, $k_{upd} = k_{stk}$, $Good_{upd} = Good_{stk}$.

- **Case 6 ($a = \rangle$) :**

This last case means that we read the end of an object. There are two possible cases.

– If the variables R, K, k and *Good* are all set to *null*, this means that we end reading an empty object $\langle\rangle$. In that case, we behave similarly to the case 5 :

$$(\mathcal{Q}, (\mathcal{Q}_{stk}, R_{stk}, K_{stk}, k_{stk}, Good_{stk}) \cdot \gamma) \xrightarrow{\rangle} (\mathcal{Q}_{upd},\gamma),$$

where $\mathcal{Q}_{upd} = \left\{ q_{upd} | \exists q \in \mathcal{Q}, q \in F^{S_j}, f(S_j) = \langle, \exists q_{stk} \in \mathcal{Q}_{stk}, q_{stk} \xrightarrow{S_j} q_{upd} \right\}$.

Finally, we update R, K, k and *Good* respectively to :

* $R_{upd} = \left\{ (p_{stk}, q_{upd}) | \exists (p_{stk}, q_{stk}) \in R_{stk}, \exists q \in \mathcal{Q}, q \in F^{S_j}, f(S_j) = \langle, q_{stk} \xrightarrow{S_j} q_{upd} \right\}$;
* $K_{upd} = K_{stk}$, $k_{upd} = k_{stk}$, $Good_{upd} = Good_{stk}$.

– In the general case, R, K, k and *Good* are not null. This means that we are ending reading an object that looks like $k_1 v_1 \#...\#kv$.

We first set the variable *Good* to $Good' = Good \cup \{(p,k,q) | (p,q) \in R\}$ (similarly to the case 4). We then compute the function $Valid(K, Good')$ (see Definition 3.2) to obtain the set of valid procedural symbols.

We can pop the top of the stack and go to the set of states $\mathcal{Q}_{upd}$, the states reachable from $\mathcal{Q}_{stk}$ (which was on top of the stack) by reading a procedural symbol from $Valid(K, Good')$.

$$(\mathcal{Q}, (\mathcal{Q}_{stk}, R_{stk}, K_{stk}, k_{stk}, Good_{stk}) \cdot \gamma) \xrightarrow{\rangle} (\mathcal{Q}_{upd}, \gamma)$$

with $\mathcal{Q}_{upd} = \{q_{upd} | \exists q_{stk} \in \mathcal{Q}_{stk}, \exists S_j \in Valid(K, Good'), q_{stk} \xrightarrow{S_j} q_{upd}\}$.
Finally, after we pop the top of the stack and go to the set of states $\mathcal{Q}_{upd}$, we can update the value of R, K, k and $Good$ respectively to :

* $R_{upd} = \{(p_{stk}, q_{upd}) | \exists (p_{stk}, q_{stk}) \in R_{stk}, \exists S_j \in Valid(K, Good'), q_{stk} \xrightarrow{S_j} q_{upd}\}$;
* $K_{upd} = K_{stk}$, $k_{upd} = k_{stk}$ and $Good_{upd} = Good_{stk}$.

The previous method doesn't apply when a VPSA reads the first and last symbol of a word w, but is pretty similar. Recall that we assume that a JSON document is always an object, thus the starting procedural automaton $\mathcal{A}^{S_0}$ of a VSPA describing a JSON schema always has $f(S_0) = \langle$. Therefore, if the word doesn't begin (resp. ends) with $\langle$ (resp. $\rangle$), it is rejected. To simplify, we assume the case $w = \langle\rangle$ as trivial.

When we read the first symbol, we look ahead to the next symbol b of the word, and we process like in case 2. We go to $\mathcal{Q} = \{p | (p, b, q) \in G^{S_0}\}$, the set of all states that can read the key b. We initialize the value of the variables to $R = \{(p, p) | p \in \mathcal{Q}\}$, $K = \{b\}$, $k = b$ and $Good = \{\}$.

Finally, when we read the last symbol of w (which should be $\rangle$), we have to check if the procedural automaton $\mathcal{A}^{S_0}$ accepts the object. We suppose that the top of the stack is empty (we reject the word if not). We then process similarly to case 6. We update $Good$ to $Good' = Good \cup \{(p, k, q) | (p, q) \in R\}$ and we compute the function $Valid(K, Good')$ (see Definition 3.2). If $Valid(K, Good') = \{S_0\}$, the word is accepted.

The complete algorithm of streaming validation of a JSON document with a VSPA using key graphs can be found in Algorithm 3.3. Note that we sometimes omit to reject the word, to increase readability of the algorithm.

Algorithm 3.3 Streaming Validation of a JSON document using a VSPA

Require : $\mathcal{A} = (\Sigma, \Gamma, \mathcal{P}, f, S_0)$ a VSPA, the set of all the key graphs G^{S_j} and a word $w = a_1 ... a_n \in \Sigma^*_{JSON}$.
Ensure : `true` is returned if $w \in L(\mathcal{A})$, `false` instead.
if $a_1 \neq \langle$ **then** **return** `false` **end if**
if $a_2 = \rangle$ **then** **return** (`true` **if** $q_0^{S_0} \in F^{S_0}$ **else** `false`) **end if**
$\mathcal{Q} \leftarrow \{p | \exists (p, a_2, q) \in G^{S_0}\}$; $\gamma \leftarrow \varepsilon$; $R \leftarrow \{(p,p) | p \in \mathcal{Q}\}$; $K \leftarrow \{a_2\}$; $k \leftarrow a_2$; $Good \leftarrow \{\}$
for $i = 2 : n-1$ **do**
  if $a_i = \sqsubset$ **then**
    $\gamma \leftarrow (\mathcal{Q}, R, K, k, Good) \cdot \gamma$; $\mathcal{Q} \leftarrow \{q_0^{S_j} | f(S_j) = \sqsubset\}$; $R, K, k, Good \leftarrow null$
  else if $a_i = \langle$ **then**
    $b \leftarrow a_{i+1}$; $\gamma \leftarrow (\mathcal{Q}, R, K, k, Good) \cdot \gamma$
    if $b = \rangle$ **then**
      $\mathcal{Q} \leftarrow \{q_0^{S_j} | f(S_j) = \langle\}$; $R, K, k, Good \leftarrow null$
    else if $b \in \Sigma_{key}$ **then**
      $\mathcal{Q} \leftarrow \{p | \exists (p, b, q) \in G^{S_j}, f(S_j) = \langle\}$
      $R \leftarrow \{(p,p) | p \in \mathcal{Q}\}$; $K \leftarrow \{b\}$; $k \leftarrow b$; $Good \leftarrow \{\}$
    end if
  else if $a_i \in \Sigma_{int} \setminus \{\#\}$ **or** ($a_i = \#$ **and** R is $null$) **then**
    $\mathcal{Q} \leftarrow \{q' | \exists q \in \mathcal{Q}, q \xrightarrow{a_i} q'\}$; $R \leftarrow \{(p,q') | \exists (p,q) \in R, q \xrightarrow{a_i} q'\}$
  else if $a_i = \#$ **then**
    $Good \leftarrow Good \cup \{(p,k,q) | (p,q) \in R\}$; $b \leftarrow a_{i+1}$;
    $\mathcal{Q} \leftarrow \{p | \exists (p, b, q) \in G^{S_j}, S_j \in \mathcal{S}\}$; $R \leftarrow \{(p,p) | p \in \mathcal{Q}\}$; $K \leftarrow K \cup \{b\}$; $k \leftarrow b$
  else if $a_i = \sqsupset$ **or** ($a_i = \rangle$ **and** R is $null$) **then**
    $(\mathcal{Q}_{stk}, R_{stk}, K_{stk}, k_{stk}, Good_{stk}) \leftarrow \gamma.pop()$
    $\mathcal{Q}_{upd} \leftarrow \{\}$; $R_{upd} \leftarrow \{\}$; $K \leftarrow K_{stk}$; $k \leftarrow k_{stk}$; $Good \leftarrow Good_{stk}$
    for $q^{S_j} \in \mathcal{Q}$ **do**
      if ($q^{S_j} \in F^{S_j}$ **and** $\overline{f(S_j)} = a_i$) **then**
        $\mathcal{Q}_{upd} \leftarrow \mathcal{Q}_{upd} \cup \left\{ q_{upd} \middle| \exists q_{stk} \in \mathcal{Q}_{stk}, q_{stk} \xrightarrow{S_j} q_{upd} \right\}$
        $R_{upd} \leftarrow R_{upd} \cup \left\{ (p_{stk}, q_{upd}) \middle| \exists (p_{stk}, q_{stk}) \in R_{stk}, p_{stk} \xrightarrow{S_j} q_{upd} \right\}$
      end if
    end for
    $\mathcal{Q} \leftarrow \mathcal{Q}_{upd}$; $R \leftarrow R_{upd}$
  else if $a_i \in \rangle$ **then**
    $Good \leftarrow Good \cup \{(p,k,q) | (p,q) \in R\}$; $(\mathcal{Q}_{stk}, R_{stk}, K_{stk}, k_{stk}, Good_{stk}) \leftarrow \gamma.pop()$
    $\mathcal{Q}_{upd} \leftarrow \{\}$; $R_{upd} \leftarrow \{\}$; $K \leftarrow K_{stk}$; $k \leftarrow k_{stk}$; $Good \leftarrow Good_{stk}$
    for $S_j \in Valid(K, Good)$ **do**
      $\mathcal{Q}_{upd} \leftarrow \mathcal{Q}_{upd} \cup \left\{ q_{upd} \middle| \exists q_{stk} \in \mathcal{Q}_{stk}, q_{stk} \xrightarrow{S_j} q_{upd} \right\}$
      $R_{upd} \leftarrow R_{upd} \cup \left\{ (p_{stk}, q_{upd}) \middle| \exists (p_{stk}, q_{stk}) \in R_{stk}, p_{stk} \xrightarrow{S_j} q_{upd} \right\}$
    end for
    $\mathcal{Q} \leftarrow \mathcal{Q}_{upd}$; $R \leftarrow R_{upd}$
  end if
end for
$Good \leftarrow Good \cup \{(p,k,q) | (p,q) \in R\}$
return (`true` **if** ($a_n = \rangle$ **and** $Valid(K, Good = \{S_0\})$ **and** $\gamma = \varepsilon$) **else** `false`)

3.3 Implementation and Experiment

We implemented the algorithms presented in this chapter in Java[1]. The actual implemented algorithms has some optimizations to improve their speed and memory usage. However, we won't expand more about these optimizations. The reader is referred to the code for more details.

In this section, we compare our algorithm with two others :

- The classical validation algorithm for JSON documents [16][2]. This algorithm recursively traverses the JSON document and the schema, checking at each node that the constraints defined in the schema are satisfied. For instance, if the current value is an object, we verify if each key is unique and as defined in the schema, and if each value paired with the key respects their corresponding schema.
- A validation algorithm using a learned *Visibly Pushdown Automata* [8][3]. A Visibly Pushdown Automaton (VPA) [3] is similar to a Finite Automaton, but with an added stack. Transitions depend on the current state, the symbol read, and the stack content. Unlike a VSPA, a VPA is not partitioned into multiple procedural automata and is not restricted in its transitions or in the elements it can push onto the stack. To help a non-initiated reader to visualize VPA, you can imagine that it is just a more complex DFA.

  This algorithm is similar to ours, as it uses a key graph and an automaton to validate JSON documents. In fact, our work on validating JSON documents is inspired by this algorithm.

Datasets

In order to make a comparison between the three algorithms, we use four JSON schemas :

- A schema that accepts an object defined recursively, that we call *Recursive List*. Each object has two keys, where the value of the first one is a string, and the value of the second one is an array containing this object schema. The abstracted schema is represented by the extended CFG with the following productions :

$$S_0 \rightarrow \langle k_1 \mathtt{s} (\varepsilon + \#k_2 S_1) \rangle$$
$$S_1 \rightarrow \sqsubset \varepsilon + S_0 \sqsupset$$

- A schema defining an object where each key-value pair corresponds to a JSON value, that is, string, number, integer, boolean, object and array. This schema is later referred to as *Basic Types*. The abstracted schema is represented by the CFG with the following productions :

$$S_0 \rightarrow \langle k_1 \mathtt{s} \#k_2 \mathtt{n} \#k_3 \mathtt{i} \#k_4 (\mathtt{true} + \mathtt{false}) \#k_5 S_1 \#k_6 S_2 \rangle$$
$$S_1 \rightarrow \langle k_7 (\mathtt{s} + \mathtt{n} + \mathtt{i} + \mathtt{true} + \mathtt{false}) \rangle$$
$$S_2 \rightarrow \sqsubset \mathtt{s} \#\mathtt{s} (\varepsilon + (\#\mathtt{s})^*) \sqsupset$$

[1] See `https://github.com/BlueTorche/VSPA`.
[2] See `https://github.com/DocSkellington/JSONSchemaTools`.
[3] See `https://github.com/DocSkellington/ValidatingJSONDocumentsWithLearnedVPA`.

- A recursive schema that defines how the metadata files for *VIM plugins* must be written[4]. The abstracted schema is represented by the CFG with the following productions (note that the usage of $(\varepsilon + \#k_i v_i)$ means that the key is optional) :

$$S_0 \rightarrow \langle k_1 \mathtt{s} \# k_2 \mathtt{s} \# k_3 \mathtt{s} \# k_4 \mathtt{s} \# k_5 \mathtt{s} \# k_6 \mathtt{s} \# k_7 S_2 \# k_8 S_1 (\varepsilon + \#\mathtt{s} U) \rangle$$
$$S_1 \rightarrow \langle \varepsilon + \mathtt{s} S_3 \rangle$$
$$S_2 \rightarrow \langle k_9 \mathtt{e} \# k_{10} \mathtt{s} \# k_{11} \mathtt{s} (\varepsilon + \#\mathtt{s} U) \rangle$$
$$S_3 \rightarrow \langle k_1 \mathtt{s} \# k_9 \mathtt{e} \# k_{10} \mathtt{s} (\varepsilon + \#k_{12} \mathtt{n}) (\varepsilon + \#k_{13} \mathtt{e}) (\varepsilon + \#\mathtt{s} U) \rangle$$

  With U being the variable producing all JSON values. We can write U with the following productions (note that these productions need to be transformed as explained in Theorem 1.3 in order to make a VSPA accepting it) :

$$U \rightarrow \mathtt{null} + \mathtt{true} + \mathtt{false} + \mathtt{i} + \mathtt{n} + \mathtt{s} + \mathtt{e}$$
$$U \rightarrow \langle \varepsilon + (\mathtt{s} U) \rangle$$
$$U \rightarrow \sqsubset \varepsilon + U (\#U)^* \sqsupset$$

- A schema that defines how *Azure Functions Proxies* files must look like[5]. As the *VIM Plugins* schema, this schema is recursive, since it uses the variable U. However, the schema of *Azure Proxies* is larger, and with a grammar that uses more variables. Therefore, the VSPA accepting the schema needs more procedural automata. The productions of the grammar are the following ones (with U the variable producing all JSON values, described earlier) :

$$S_0 \rightarrow \langle k_1 \mathtt{s} \# k_2 S_1 \rangle$$
$$S_1 \rightarrow \langle k_3 S_2 (\varepsilon + \mathtt{s} S_2) \rangle$$
$$S_2 \rightarrow \langle k_4 S_3 \# k_5 S_4 \# k_6 \mathtt{s} \# k_7 S_5 \# k_8 S_6 \# k_9 (\mathtt{true} + \mathtt{false}) \# k_{10} (\mathtt{true} + \mathtt{false}) \rangle$$
$$S_3 \rightarrow \sqsubset \varepsilon + \mathtt{s} (\#\mathtt{s})^* \sqsupset$$
$$S_4 \rightarrow \langle k_{11} S_7 \# k_{12} \mathtt{s} \rangle$$
$$S_5 \rightarrow \langle k_{13} U \# k_{14} \mathtt{s} \# k_{15} \mathtt{s} (\varepsilon + \#k_{16} \mathtt{s}) (\varepsilon + \#k_{17} \mathtt{s}) \rangle$$
$$S_6 \rightarrow \langle k_{18} U \# k_{19} \mathtt{s} \# k_{20} \mathtt{s} \# k_{21} (\mathtt{s} + S_8 + S_9) (\varepsilon + \#k_{22} \mathtt{s}) \rangle$$
$$S_7 \rightarrow \sqsubset \mathtt{e} (\#\mathtt{e})^* \sqsupset$$
$$S_8 \rightarrow \sqsubset S_9 (\#S_9)^* \sqsupset$$
$$S_9 \rightarrow \langle \varepsilon + (\mathtt{s} U) \rangle$$

Note that, in our datasets, we only have two schemas that are used as real use cases. This is due to the fact that we must manually build the CFG and the VSPA accepting the schema, which takes time and can lead to mistakes. Section 4.1.1 discusses how we can automatically build a VSPA accepting a JSON schema.

[4] See `https://json.schemastore.org/vim-addon-info.json`.
[5] See `https://json.schemastore.org/proxies.json`

Based on these schemas, we generate valid and invalid JSON documents using a random generator [8]. This generator follows the schema to create documents, making random choices when there are disjunctions, and may deviate from the schema to create an invalid document.

For each schema, we generate 10,000 JSON documents with variable length. The maximum depth of recursive documents is 20. Note that 20 to 40% of documents are invalid, to assess whether the algorithms properly validate the schema.

Comparison of the Automata

VSPAs and VPAs are both extensions of automata. In the case of VPA, the automaton is obtained through a *learning process*, meaning that it is build automatically (we discuss a bit more of the learning process in Section 4.1.1). The resulting VPA is *minimal* for the class of VPA that is learned [2, 8], meaning that its number of states is the lowest possible.

First, we compare the obtained automata in terms of the number of states and transitions. The comparison of the automata can be found in Table 3.1. In this table, $|\mathcal{P}|$ represents the number of procedural automata, $|Q|$ the number of states, and $|\delta|$ the number of transitions. In the case of VSPAs, the number of states and transitions are the sum of all the procedural automata. In the case of VPAs, the number of transitions are only the *useful* ones, meaning that transitions which would immediately cause the word to be rejected are excluded.

We observe that, for complex schemas, the VSPA have approximately half as many states than the VPA. This isn't surprising, for two reasons. First, the procedural automata of the VSPA are non-deterministic, allowing them to have fewer states. The theoretical bound of the number of states for a deterministic procedural automaton would be $\mathcal{O}(2^{|Q|})$ (see Appendix A). However, if we manually construct deterministic procedural automata accepting the schema, the number of states would be roughly the same. Secondly, VSPA are more *natural* to describe a JSON schema. The procedural automaton obtained are linear, as the regular expression describing the schema are simple. This leads to more clarity to understand the automata, and fewer states than VSPA.

On the other hand, we observe that VSPA have significantly fewer transitions than VPA. This explains itself due to the large number of return transitions in VPA, which constitute the majority of the transitions in the learned VPA [8]. In contrast, VSPAs only have internal and procedural transition. Note that, even if the procedural automata would be deterministic, the number of transitions would be significatively fewer than for VPA (if we exclude transitions which would immediately cause the word to be rejected). Note also that the learned VPA is minimal in terms of the number of states, but not in terms of the number of transitions.

	VSPA			**VPA**	
Dataset	$\lvert\mathcal{P}\rvert$	$\lvert Q\rvert$	$\lvert\delta\rvert$	$\lvert Q\rvert$	$\lvert\delta\rvert$
Recursive List	2	8	6	7	22
Basic Types	3	26	30	24	77
VIM plugins	6	67	115	150	4007
Azure Proxies	11	74	121	121	3117

Table 3.1: Comparison of the size of the VSPA and VPA [8].

Then, we compare the obtained key graphs for VSPAs and VPAs. Both key graphs are automatically computed from the automata, but the algorithms used are different, as some adaptations need to be made to detect paths reading key value pairs in a VPA [8]. Table 3.2 shows the comparison of computation time and memory usage of the key graphs, as well as their number of vertices. For VSPA, the reported value represents the cumulative computation time and memory usage for all key graphs, averaged over 100 runs.

The first thing we notice is that the computation time for VSPAs is negligible – on the order of 10-100µs – compared to that of the VPAs. This is easy to explain. For a VSPA, only a simple graph exploration is required to compute the key graph, which makes it fast to compute. However, for a VPA, many stacked runs are generated to compute its key graph. Furthermore, the theoretical time complexity of computing the key graph for a VPA [8] is higher than that of the VSPA (see (3.1)), even when cumulating the time complexity for all the procedural automata describing an object schema.

Another observation is that, for *VIM plugin* and *Azure Proxies* schemas, the size of the key graphs generated by the VSPA is drastically smaller than that of the VPA. This can be explained by the number of transitions in the VPA. As previously discussed, VPAs have significantly more transitions than VSPAs. This leads to a greater number of possible paths for reading a key value pair, which results in more vertices in the key graph. In contrast, the VSPAs are more compact, with only a few possible paths to read a key value pair. This results in fewer vertices in their key graphs. In fact, we observe that there is only one vertex per occurrence of each key in the grammar describing the schema.

Obviously, the memory used by the VSPA to compute the key graphs is also significantly smaller than the memory used by VPA. This is logical, as it takes less time to compute, needs to check fewer paths and results in fewer states. Note that some inaccuracies may have occurred in memory measurement for the VSPA. We discuss later these possible mistakes, but we suspect VSPAs to use less memory than those presented in this table.

	VSPA			**VPA**		
Dataset	**Time (ms)**	**Memory (kB)**	**Size**	**Time (ms)**	**Memory (kB)**	**Size**
Recursive List	$\ll 1$	61	2	55	2095	3
Basic Types	$\ll 1$	61	7	63	2164	9
VIM plugins	$\ll 1$	61	21	184	2315	306
Azure Proxies	$\ll 1$	62	24	150	976	541

Table 3.2: Comparison of the computation time and memory usage to compute the Key Graphs, and their size, for VSPA and VPA.

Validation Results

For each schema, we execute the algorithms on 10,000 JSON documents. To avoid measurements interfering with each other, we first execute the algorithms while measuring the time, then we execute them while measuring the memory. For VSPAs, the execution time is averaged over 100 runs, and for VPAs and the classical validator, it is averaged over 10 runs. Note that the three algorithms return the same validation output on all documents (true if the document belongs to the schema, false otherwise).

Figures 3.3 and 3.4 show the results for the *VIM Plugins* and *Azure Proxies* schemas. Green squares represent the values for the VSPA, blue crosses for the VPA, and red circles for the classical algorithm. The x-axis of the graphs represents the number of symbols in each document. Results of *Recursive List* and *Basic Types* schemas are less relevant to discuss, but are provided in Appendix E.

We first compare the computation time required by each algorithm to validate documents (see Figures 3.3a and 3.4a).

For the *VIM Plugins* schema, we observe that the classical algorithm usually takes less than one millisecond. The VPSA and VPA algorithms scale proportionally with the document length, with VSPA being slightly faster.

For *Azure Proxies*, we observe similar things, except for the execution time of the classical algorithm. We see that it takes several milliseconds to process some documents, which appear to be the valid ones. This may be caused by the presence of some disjunctions in the schema, which take time to process for the classical algorithm. For small valid documents, the computation time is significatively higher for the classical algorithm than for VSPAs and VPAs.

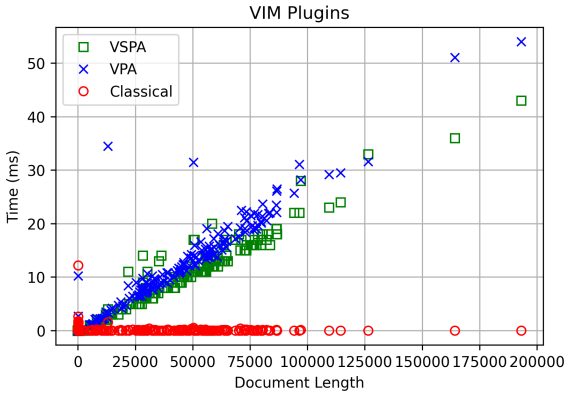
Note that the VSPA validation algorithm may not be as optimized as the other two algorithms. This may result in increased computation time when validating some documents.

Before comparing the memory usage, note the absence of the classical validator in the figures corresponding to memory usage. Since it is not a streaming algorithm, it must load the entire document into memory in order to validate it. Therefore, the classical algorithm needs way more memory to compute large documents than VSPAs and VPAs. For instance, the memory usage of the three algorithms for the *VIM Plugins* schema is given in Figure 3.5.

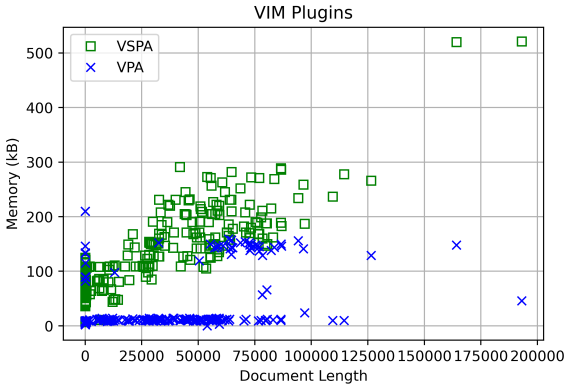
The memory usage to validate the JSON documents can be found in Figures 3.3b and 3.4b[6]. We can observe that the memory usage by VSPAs is generally higher than that of VPAs. For both schemas, the memory usage of VSPAs is usually around 10 times higher than that of VPAs. It seems logical that a VSPA uses more memory, as it is non-deterministic, it may store many configurations at the same time. This may drastically increase memory usage.

However, memory measurement for VSPAs may have some issues. Upon analysis, we observed irregularities that should not have occurred, especially on small documents. For instance, when we repetitively run the memory measurement of the validation, we may have totally different results. Some implementation issues may have introduced some noise or errors in the measurement. Although we think that VSPAs use more memory than VPAs, we think that it uses less memory than what is reported here.

[6]For *Azure Proxies*, the memory usage was measure only for 8,000 documents.

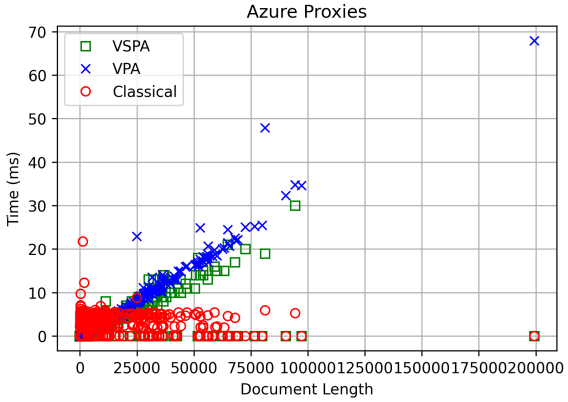


(a) Computation time depending on the number of symbols in the document.

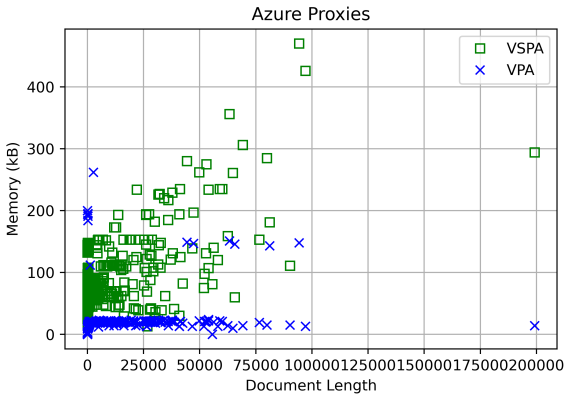


(b) Memory usage depending of the number of symbols in the document.

Figure 3.3: Time and Memory Results on validation of JSON documents of *VIM plugin* schema. Green squares correspond to the VSPA algorithm, blue crosses to the VPA algorithm, and red circles to the classical algorithm.



(a) Computation time depending on the number of symbols in the document.



(b) Memory usage depending on the number of symbols in the document.

Figure 3.4: Time and Memory Results of validation of JSON documents of *Azure Proxies* schema. Green squares correspond to the VSPA algorithm, blue crosses to the VPA algorithm, and red circles to the classical algorithm.

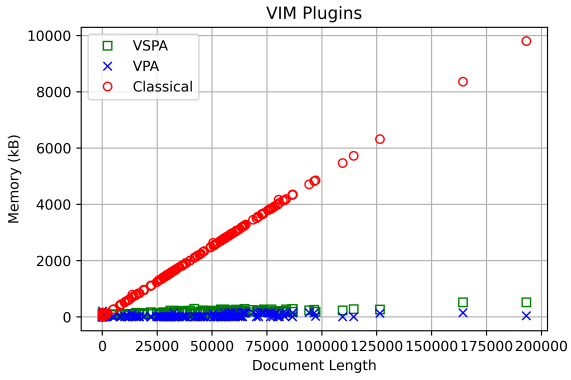


Figure 3.5: Memory usage of the validation of JSON documents of *VIM Plugins* for the three algorithms. Green squares correspond to the VSPA algorithm, blue crosses to the VPA algorithm, and red circles to the classical algorithm.

Table 3.3 provides a summary of the results for each schema. This table includes the average computation time and memory usage per symbol in the JSON document. The table also shows the worst-case computation time and memory usage observed.

We observe that the average memory usage for VSPA is relatively high, in the same order of magnitude than the classical algorithm. This is due to the irregularities on small documents, as previously discussed. As the memory usage on small documents (which constitute the majority of our dataset) is high, this creates a high memory usage per symbol.

Finally, we observe that the average computation time of VSPAs is slightly lower than that of VPAs, as already seen in the previous figure. The average time of the classical algorithm for *Azure Proxies* is high, due to the execution time required to process small valid documents.

Note that, for VPAs and the classical algorithm, the computation time measurement is done in millisecond. This leads to a low precision on the mean time in Table 3.3 when the algorithm takes less than one millisecond to validate most of the documents.

	Basic Types			
Algorithm	**Mean Time (µs/symbol)**	**Mean Memory (kB/symbol)**	**Max Time (ms)**	**Max Memory (kB)**
VSPA	0.059	0.32	0.45	453
VPA	—[7]	0.023	4	100
Classical	—[7]	0.10	3	8384
	Recursive List			
VSPA	0.014	0.35	0.62	829
VPA	—[7]	0.009	3	71
Classical	—[7]	0.07	3	7180
	VIM Plugins			
VSPA	0.25	0.050	44	510
VPA	0.27	0.006	54	201
Classical	0.032[7]	0.059	12	9804
	Azure Proxies			
VSPA	0.25	0.026	21	470
VPA	0.32	0.0044	68	262
Classical	1.2	0.087	22	9960

Table 3.3: Average computation time and memory usage and worse case of computation time and memory usage for each algorithm and each schema.

[7] Low precision due to time measurement in millisecond.

Chapter 4

Conclusion

4.1 Future Work

4.1.1 Automatically Built VSPA

In this work, we had to manually model the VSPA that accepts a JSON schema. This process is time-consuming and lead to mistakes. We had to verify whether the validation of each document was correct by using the classical algorithm, to ensure that the constructed VSPA was correct. In this section, we discuss how a VSPA could be built automatically.

From the schema

First, we could automatically build the VSPA directly from the schema. As briefly seen in section 1.4.2, there are a limited number of constraints that a JSON schema can apply. It should be easy to build an NFA directly from these constraints, and obtain a VSPA accepting the schema. Furthermore, it should be easier to construct a VSPA from the schema than to construct a VPA from the schema.

However, creating a VSPA directly from the schema may result in unnecessary states in its procedural automata if the schema is poorly designed. Additionally, this would work only in the use case of JSON schema, and we could not apply this method to other types of documents, such as XML files.

Active Learning Algorithm

It is possible for some types of automata to be built using an *active learning algorithm*. For instance, given a regular language L, it is possible to create an algorithm that construct a DFA $\mathcal{A}$ such that $L(\mathcal{A}) = L$ using the *Angluin framework* [4]. This works by querying a *teacher*, that knows the language L. An active learning algorithm also exists for VPA [8], which was used in this work to construct the VPAs accepting the JSON schemas.

This has some advantages, as it is not directly learned from the schema, it is not affected by some potential errors in the schema [19]. Furthermore, an active learning algorithm can have applied to any kind of language, and is not limited to the application of JSON.

Frohme already created an active learning algorithm for SPA [10]. However, due to the non-determinism we have introduced with the surjective function f, it is not possible to apply this algorithm to VSPA. The main problem in the learning is that a word may belong to the language of two procedural automata, and it becomes hard to compare the language of two procedural automata.

To address this, we propose splitting each procedural automaton whenever it has more than one final state. The resulting procedural automata would be similar to the original one, but with some states no longer final, to ensure a unique final state. We can also remove bin states in the resulting automata to make them easier to compare. This can ensure that no words belong to two procedural automata.

By doing this, we could potentially build a bottom-up algorithm that starts with the deepest nested part of the word, and ends with the starting procedural automaton. Finally, after that the VSPA was learned, if some procedural automata are always used together – that is, $\delta(q,J) = \delta(q,K)$ for all q – we can merge them.

Another possibility of learning algorithm would consist in build a unique procedural automaton for each call symbol (for instance, in case of JSON, two procedural automata, one for brackets and one for braces). We then "mark" the final states of these procedural automata to identify when they can be used in a transition in the VSPA. Finally, we split the procedural automata depending on the possible transitions, and we remove the bin states.

We have not worked yet to formally define this algorithm, and we obviously have not proven that the algorithm would work. However, this could serve as a good foundation to build an active learning algorithm for VPSA.

4.1.2 ε-SPA

In Section 1.2.2, we defined NFAs (see Definition 1.5) as automata that may have multiple transitions from a single state for the same symbol. Theoretically, an NFA can even include ε-transitions, meaning that it can change its current state without reading any symbol (that is, by reading the empty word ε) [12]. Note that this does not affect the conclusion of Theorem 1.2, nor the construction provided in Appendix A.

A similar concept could be imagined for VSPAs. In this variant, which we refer to as an ε-SPA, the calls and returns of a procedure are performed by reading the empty word ε. This means that both the call and return alphabet are empty, and the linking function f maps each procedural symbol to ε. With such a definition, we could construct an ε-SPA accepting any CFG.

Naturally, this would introduce additional nondeterminism to the automaton. In fact, it would likely be impossible to design a streaming validation algorithm for most CFGs. However, such an algorithm might still be feasible for certain subclasses of CFGs. It may even be possible to design an active learning algorithm of an ε-SPA for these CFGs.

4.1.3 Equivalence of Class of Language

It would be interesting to determine the class of languages accepted by VSPAs. Let SPL denote the set of languages accepted by an SPA, VSPL the set accepted by a VSPA, VPL the set accepted by a VPA, and ε-SPL the set accepted by an ε-SPL, we conjecture the following strict inclusions :

$$\text{SPL} \subsetneq \text{VSPL} \subsetneq \text{VPL} \subsetneq \varepsilon\text{-SPL} \equiv \text{CFL}.$$

This conjecture does not appear to be difficult to prove. We propose here some foundation to prove it.

Any SPA can be represented by a VSPA in which the linking function f is bijective. However, when the function f is surjective, there exists no equivalent SPA that accepts the same language.

Then, if we adapt the validation algorithm described in Section 3.1.1, it is likely possible to construct a VPA from a given VSPA. However, VPAs can accept any regular expression, but VSPAs cannot. By Definition 2.6, any word accepted by a VSPA must contain at least two symbols. Even if we modify this definition, we could still construct a language that is not accepted by any VSPA. However, by modifying the semantics of return rules of a VSPA, we might be able to obtain an equivalent of VPAs.

It is already proven that VPA$\subsetneq$CFL [3]. To prove that ε-SPL$\equiv$CFL, a formal definition of ε-SPL would be required.

4.2 Other Use Cases for VSPAs

A VSPA can be applied to any tag-based language – such as XML or HTML – which is structured through observable entry and exit points. In fact, SPA can already be used for such languages. Beyond these, a VSPA can also be used for any language recursively structured using delimiter, such as JSON documents.

Moreover, most network transmissions could be modeled as a tag-like language, since they typically include flags to identify the type of message. Using these flags, it could be possible to quickly verify the transmission received with a VSPA. Regular expressions are commonly used in static malware analysis [18], usually implemented with finite automata. We could imagine a tool that scans large volumes of transmissions using a VSPA, where some procedural automata are dedicated to malware detection.

Automata are also widely used in model checking [5]. For complex systems, it is often impossible to guarantee the complete absence of errors. However, if the system can be modeled as an automaton, verifying the model becomes a feasible task. The key challenge is to build a good model that accurately reflects the behavior of the original system. The availability of different types of automata that can be learned facilitates the construction of such accurate models.

4.3 Conclusion

In this work, we presented the Visibly System of Procedural Automata (VSPA), an extension of finite automata that can mutually call each other. We introduced a mechanism to handle the nondeterminism of the call procedures. This model enables the validation of documents with recursive structure and delimiters. As an application, we choose to validate a set of JSON documents against a JSON schema. We proved that a VSPA can be constructed to accept the same language defined by the JSON schema.

We then introduced the key graph for procedural automata, which makes it possible to handle JSON objects with unordered of key-value pairs. Using the key graph, we developed a streaming validation algorithm. Finally, we compared our algorithm to existing ones, to show that it can be efficient in certain cases.

Bibliography

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *Automata, Languages and Programming (ICALP 2005)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1102–1114. Springer, 2005.

[3] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 202–211, New York, NY, USA, 2004. Association for Computing Machinery.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, 2008.

[6] Noel Benji. Designing scalable json schemas for computer vision pipelines, March 2025. Published on Medium.

[7] Tim Bray. The javascript object notation (json) data interchange format. RFC 8259, 2017.

[8] Véronique Bruyère, Guillermo A. Pérez, and Gaëtan Staquet. *Validating Streaming JSON Documents with Learned VPAs*, page 271–289. Springer Nature Switzerland, 2023.

[9] Shimon Even. *Graph Algorithms*. Cambridge University Press, 2nd edition, 2011.

[10] M. Frohme and B. Steffen. Compositional learning of mutually recursive procedural systems. *International Journal on Software Tools for Technology Transfer*, 23:521–543, 2021.

[11] Viliam Geffert. Formal models and semantics. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 235–271. MIT Press, 1990.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1st edition, 1979.

[13] JSON-Schema.org. JSON-Schema.org website. `https://json-schema.org/`. Online, accessed: 2025.

[14] JSON.org. JSON.org website. `https://www.json.org`. Online, accessed: 2025.

[15] Object Management Group. *Unified Modeling Language (UML) 2.5.1 Specification*, chapter StateMachines, page 305. Object Management Group (OMG), December 2017. OMG Document Number: formal/2017-12-05.

[16] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. Foundations of json schema. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web (WWW 2016)*, pages 263–273, Montreal, Canada, 2016. ACM.

[17] Juraj Sebej. Reversal of regular languages and state complexity. In *Conference on Theory and Practice of Information Technologies*, 2010.

[18] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, 2012.

[19] Claire Yannou-Medrala and Fabien Coelho. An Analysis of Defects in Public JSON Schemas. In *39èmes journées de la conférence BDA Gestion de Données – Principes, Technologies et Applications*, Montpellier, France, October 2023.

Appendix A

Equivalence of NFAs and DFAs

Theorem 1.2 says that, for any NFA $\mathcal{A}$, there exists a DFA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$, and conversely. In this appendix, we develop a construction of a DFA $\mathcal{B}$ accepting the same language as a given NFA $\mathcal{A}$.

Let $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ be an NFA. We can construct a DFA $\mathcal{B} = (\Sigma', Q', q_0', F', \delta')$ accepting the same language as $\mathcal{A}$. This DFA is defined as :

- $\Sigma' = \Sigma$;
- $Q' = 2^Q$, with 2^Q the set of all subsets of Q : $2^Q = \{P | P \subseteq Q\}$. Note that $|2^Q| = 2^{|Q|}$.
- $q_0' = \{q_0\}$;
- $F' = \{P \in Q' | P \cap F \neq \emptyset\}$.
- $\delta' : Q' \times \Sigma \to Q'$: $\delta'(P,a) = \bigcup_{q \in P} \delta(q,a)$, with $P \in Q'$ and $a \in \Sigma$.

By this definition, we note that a state $P \in Q'$ is a final state of $\mathcal{B}$ if it contains at least one final state of $\mathcal{A}$. We can also note that, for $P, P' \in Q'$ and $a \in \Sigma$, a transition $P \xrightarrow{a} P'$ exists in δ' if and only if, for all $q' \in P'$, it exists a state $q \in P$ such that $q \xrightarrow{a} q' \in \delta$.

The proof that, for a given NFA $\mathcal{A}$, the DFA $\mathcal{B}$ obtained with this construction accepts the same language $L(\mathcal{B}) = L(\mathcal{A})$ is formally demonstrated in [12].

Let's take the 3-state NFA $\mathcal{A}$ shown in Figure 1. This NFA accepts the language $L(\mathcal{A}) = \{w \in \{a,b\}^* | bb \in Suff(w)\}$.

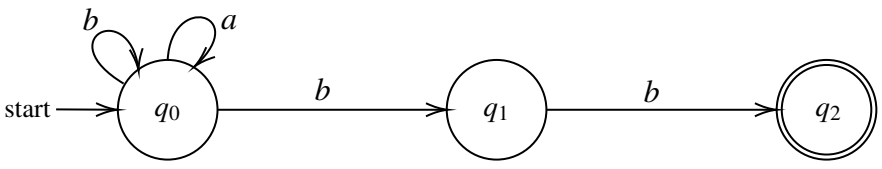


Figure 1: NFA $\mathcal{A}$ accepting the language $L(\mathcal{A}) = \{w \in \{a,b\}^* | bb \in Suff(w)\}$.

We construct a DFA $\mathcal{B}$ accepting the same language. This DFA has $2^3 = 8$ states. Here is the exhaustive list of all states in Q' : $\emptyset$, $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0,q_1\}$, $\{q_0,q_2\}$, $\{q_1,q_2\}$, $\{q_0,q_1,q_2\}$. The transition function δ' of $\mathcal{B}$ is constructed as described in the previous construction. Figure 2 describes this transition function. The resulting oriented graph representing the DFA $\mathcal{B}$ can be viewed in Figure 3.

δ'		**Symbol** a	b
State	$\emptyset$	$\emptyset$	$\emptyset$
	$\{q_0\}$	$\{q_0\}$	$\{q_0,q_1\}$
	$\{q_1\}$	$\emptyset$	$\{q_2\}$
	$\{q_2\}$	$\emptyset$	$\emptyset$
	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0,q_1,q_2\}$
	$\{q_0,q_2\}$	$\{q_0\}$	$\{q_0,q_1\}$
	$\{q_1,q_2\}$	$\emptyset$	$\{q_2\}$
	$\{q_0,q_1,q_2\}$	$\{q_0\}$	$\{q_0,q_1,q_2\}$

Figure 2: Transition Function δ' of DFA $\mathcal{B}$.

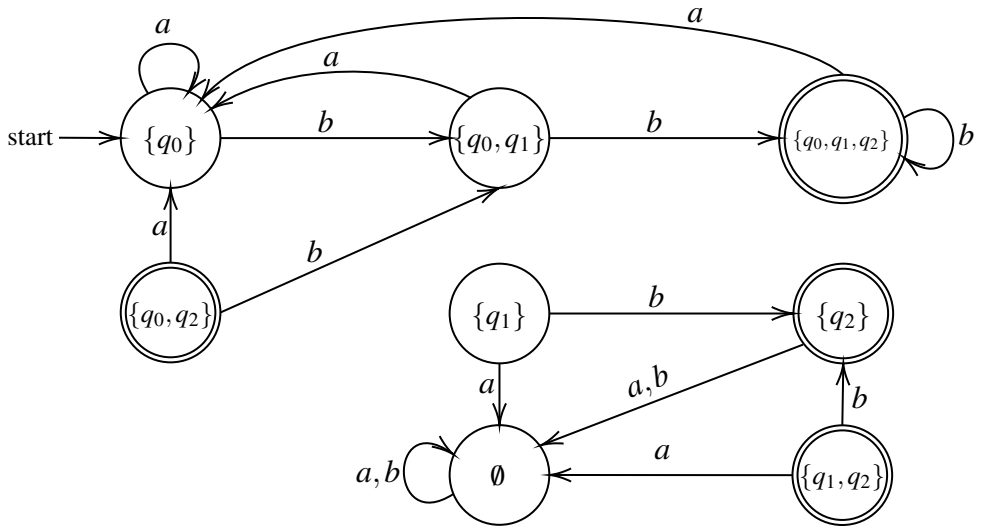


Figure 3: DFA $\mathcal{B}$ constructed to accept the same language as NFA $\mathcal{A}$ (see Figure 1).

We can note that, in the 8 states of $\mathcal{B}$, only 3 states are reachable from the initial state $\{q_0\}$. Removing the 5 unreachable states of $\mathcal{B}$ will result in a DFA that accepts the same language.

If we focus on the three reachable states $\{q_0\}$, $\{q_0,q_1\}$ and $\{q_0,q_1,q_2\}$, it's easy to see that the language accepted by $\mathcal{B}$ contains all words ending by bb, which is the language $L(\mathcal{A})$.

You can note that the formal construction always gives a DFA of $2^{|Q|}$ states. In practice, most of the NFAs can be represented as a DFA with less than $2^{|Q|}$ states. A good way to find such a DFA is to start the construction of the DFA from the initial state $\{q_0\}$, and only focus on reachable states.

However, we can show that there exist regular languages that, if the NFA with the fewest states accepting it have n states, then there exists no DFA accepting it that have less than 2^n states [17].

Appendix B

Equivalence of the Class of Languages of CFG and Extended CFG

Theorem .1. *Let L be a language.* $\exists \mathcal{G}$ *a CFG such that* $L(\mathcal{G}) = L \iff \exists \mathcal{G}'$ *an extended CFG such that* $L(\mathcal{G}') = L$.

Proof :

- $\exists \mathcal{G}$ a CFG with $L = L(\mathcal{G}) \Rightarrow \exists \mathcal{G}'$ an extended CFG with $L = L(\mathcal{G}')$:

  This part is trivial. The difference between a CFG and an extended CFG is the right-hand side of the production, noted α. For a CFG, α can only be a **word** over $(V \cup T)$. For an extended CFG, α can be a **regular expression** over $(V \cup T)$. However, any word over an alphabet is a regular expression over this same alphabet (see Definition 1.4). Therefore, a CFG is also an extended CFG.

- $\exists \mathcal{G}$ a CFG with $L = L(\mathcal{G}) \Leftarrow \exists \mathcal{G}'$ an extended CFG with $L = L(\mathcal{G}')$:

  This part isn't trivial, still, it makes sense that any extended CFG can be represented as a CFG. An extended CFG uses regular expression over $\Sigma = V \cup T$. But any regular expression over an alphabet Σ can be represented by a CFG over an alphabet $\Sigma' = V' \cup \Sigma$. Intuitively, it seems possible to build a CFG representing the same language as an extended CFG by adding some variables and/or productions.

  If we take Definition 1.4 of regular expressions, we show that any productions $S \to l$, with l a regular expression over an alphabet, can be represented by productions of a CFG. The regular expressions representing the empty word and a single symbol are trivial (as they are word, they are already a correct production for a CFG) and are the base cases. Furthermore, an extended CFG $\mathcal{G}$ with $L(\mathcal{G}) = \emptyset$ can be represented as a CFG with an empty production set P. By induction, we prove that all productions of extended CFG can be represented by productions of a CFG :

1. $S \to l_1 + l_2$, with l_1 and l_2 regular expressions, denoting the regular language $L_1 \cup L_2$:
   Here are the productions representing the regular expression $l_1 + l_2$ with $\{S\}$ as the set of variables :

$$S \to l_1,$$
$$S \to l_2.$$

   By induction, l_1 and l_2 can be represented by production using only words. Then, $S \to l_1 + l_2$ can be represented by productions of a CFG.

2. $S \to l_1 l_2$, with l_1 and l_2 regular expressions, denoting the regular language $L_1 L_2 = \{u \cdot v | u \in L_1, v \in L_2\}$:
   Here are the productions producing the regular expression $l_1 l_2$ with $\{S, A, B\}$ as the set of variables :

$$S \to AB,$$
$$A \to l_1,$$
$$B \to l_2.$$

   By induction, l_1 and l_2 can be represented by productions using only words. Then, $S \to l_1 l_2$ can be represented by productions of a CFG.

3. $S \to l^*$, with l a regular expression, denoting the regular language L^* :
   Here are the productions producing l^* with $\{S\}$ as the set of variables :

$$S \to Sl,$$
$$S \to \varepsilon.$$

   By induction, l can be represented by production using only words. Then, $S \to Sl$ can be represented by production of a CFG, and $S \to l^*$ too.

With these steps, by adding some variables and productions, we can transform any production $S \to l$ with l a regular expression in a set of productions of the form $V_i \to \alpha$, with α a word over the alphabet of variables and terminals.

As an example, we will transform the extended CFG $\mathcal{G}$ given in Section 1.3.2 into a CFG $\mathcal{G}'$. As a recall, we defined $\mathcal{G} = (\{S, R\}, \{a, b, z\}, P, S)$ an extended CFG, where P is the set of the following productions :

$$\begin{aligned} S &\to a(R + S)^* z \\ R &\to abz. \end{aligned}$$

We define the CFG $\mathcal{G}' = (\{S,R,A,B\},\{a,b,z\},P',S)$ such that $L(\mathcal{G}') = L(\mathcal{G})$, where P' is the set of following productions :

$$\begin{aligned} S &\rightarrow aAz, \\ A &\rightarrow AB, \\ A &\rightarrow \varepsilon, \\ B &\rightarrow S, \\ B &\rightarrow R, \\ R &\rightarrow abz. \end{aligned}$$

The CFG $\mathcal{G}'$ has been constructed by following the rules of construction in the previous proof. As you can note, a CFG may have a lot more productions than its corresponding extended CFG.

F

Appendix C

Example of Transformation of a JSON Schema According to Theorem 1.3

The JSON Schema in Figure 1.4 is represented by an extended CFG $\mathcal{G} = (\mathcal{S}, \Sigma_{JSON}, P, S_0)$, with $\mathcal{S} = \{S_0, S_1, S_2, S_3, S_4\}$, $\Sigma_{JSON} = \Sigma_{pVal} \cup \Sigma_{key} \cup \{\#, \langle, \rangle, \sqsubset, \sqsupset\}$, $\Sigma_{key} = \{k_1, k_2, k_3, k_4, k_5\}$ such that k_1 is the symbol representing `title`, $k_2 \equiv$ `conference`, $k_3 \equiv$ `keywords`, $k_4 \equiv$ `name`, $k_5 \equiv$ `year`, and the set of productions are defined by :

$$
\begin{aligned}
S_0 &\rightarrow \langle k_1 S_1 \# k_2 S_2 \# k_3 S_3 \rangle \\
S_1 &\rightarrow \texttt{s} \\
S_2 &\rightarrow \sqsubset \varepsilon + (S_1 (\# S_1)^*) \sqsupset \\
S_3 &\rightarrow \langle k_4 S_4 \# k_5 S_5 \rangle \\
S_3 &\rightarrow \langle k_4 S_4 \rangle \\
S_3 &\rightarrow \langle k_5 S_5 \rangle \\
S_3 &\rightarrow \langle \varepsilon \rangle \\
S_4 &\rightarrow \texttt{n} \\
S_5 &\rightarrow \texttt{i}
\end{aligned}
$$

where we add to S_0 and S_3 all the productions mandatory to *close* the extended CFG $\mathcal{G}$.

Theorem 1.3 say that we can construct an extended CFG $\mathcal{G}' = (\mathcal{S}, \Sigma_{JSON}, P', S_0)$ describing the same schema $L(\mathcal{G}) = L(\mathcal{G}')$. To construct $\mathcal{G}'$, we first need to write all the productions with left-hand sides S_3 as a unique production. We then remove production with left-hand sides S_1, S_4 and S_5, as they are primitive schemas, and not of the form $S_i \rightarrow a_i \alpha_i \bar{a}_i$. Finally, we write the boolean expression in S_3 as a regular expression. The set of productions of $\mathcal{G}'$ is :

$$
\begin{aligned}
S_0 &\rightarrow \langle k_1 \texttt{s} \# k_2 S_2 \# k_3 S_3 \rangle \\
S_2 &\rightarrow \sqsubset \varepsilon + (\texttt{s} (\# \texttt{s})^*) \sqsupset \\
S_3 &\rightarrow \langle (k_4 \texttt{n} \# k_5 \texttt{i}) + (k_4 \texttt{n}) + (k_5 \texttt{i}) + \varepsilon \rangle
\end{aligned}
$$

To close the extended CFG $\mathcal{G}'$, as we cannot add productions, we need to *close* the regular expressions describing S_0 and S_3. This is easy to do, but it increases exponentially the size of the regular expressions :

$$\begin{aligned}
S_0 \rightarrow & \langle (k_1\mathtt{s}\#k_2S_2\#k_3S_3) + (k_1\mathtt{s}\#k_3S_3\#k_2S_2) + (k_2S_2\#k_1\mathtt{s}\#k_3S_3) + \\
& (k_2S_2\#k_3S_3\#k_1\mathtt{s}) + (k_3S_3\#k_2S_2\#k_1\mathtt{s}) + (k_3S_3\#k_1\mathtt{s}\#k_2S_2)\rangle \\
S_2 \rightarrow & \sqsubset \varepsilon + (\mathtt{s}(\#\mathtt{s})^*) \sqsupset \\
S_3 \rightarrow & \langle (k_4\mathtt{n}\#k_5\mathtt{i}) + (k_5\mathtt{i}\#k_4\mathtt{n}) + (k_4\mathtt{n}) + (k_5\mathtt{i}) + \varepsilon\rangle.
\end{aligned}$$

We usually omit to explicitly close productions over objects to improve readability.

Appendix D

Proofs of Properties of Words accepted by a VSPA

These section proves the properties presented in Section 2.1.3. As a reader, you may notice a certain lack of precision in the proofs of Properties (2.4) and (2.8). This informality is intentional, aiming to make the work clearer and more concise by avoiding the introduction of additional technical tools.

Proof of Property (2.4)

Let w a word accepted by a procedural automaton $\mathcal{A}^J$, w is a well-matched word :

$$w \in L(\mathcal{A}^J) \Rightarrow w \in WM(\Sigma),$$
$$L(\mathcal{A}^J) \subseteq WM(\Sigma).$$

Let $aw\overline{a} \in L(\mathcal{A}^J)$ a word over the alphabet $\Sigma = \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$ accepted by the procedural automaton $\mathcal{A}^J$, with $a = f(J)$. By Definition 2.7 of well-matched word, $aw\overline{a}$ is well-matched if w is well-matched.

First, it is easy to prove that w respects Properties (2.1), (2.2) and (2.3) (but they are not sufficient to prove that w is well-matched if $|\Sigma_{call}| > 1$). If w would have one of its prefixes that has more return symbols than call symbols, that means that, at some point of its reading, we need to pop a state from an empty stack. This isn't allowed by the semantics of VSPA (see Definition 2.4, and the word would be rejected. Furthermore, as the stacked run begins and ends with an empty stack, w should have as many call and return symbols, as we stack an item when we read a symbol, and we pop an item when we read a return. Therefore, Properties (2.1) and (2.2) are respected, which imply Property (2.3)

Then, we proceed by induction over the depth of w.

Base case $depth(w) = 0$ **:**

As the depth of w is 0, by Definition 2.9, for all prefixes u of w, $\beta(u) \leq 0$. However, w respect (2.2) : $\beta(u) \geq 0$. Therefore, $\beta(u) = 0$ for all u prefixes of w. From that point, it's easy to see no prefix u have a call or a return symbol, thus $w \in \Sigma_{int}^*$. By Definition 2.7, w is a well-matched word.

Induction step $depth(w) > 0$ **:**

As w respects Properties (2.1), (2.2) and (2.3), w is of the form :

$$w = u_0 a_1 w_1 b_1 u_1 a_2 w_2 b_2 ... u_{m-1} a_m w_m b_m u_m,$$

with $u_i \in \Sigma_{int}^*$, $a_i \in \Sigma_{call}$, $b_i \in \Sigma_{return}$ and $w_i \in \Sigma^*$ (and w_i respects Properties (2.1), (2.2) and (2.3)), for all i and for some $m \in \mathbb{Z} \setminus \{0\}$.

It is easy to see that $depth(w_1) < depth(w)$, as we removed the call symbol a_1 from the prefixes of w_1. If we look at the call and return rules (see Figure 2.3), since w_i respects Properties (2.1), (2.2) and (2.3), we see instinctively that it exists the following stacked run :

$$(q, \varepsilon) \xrightarrow{a_1} (q_0^J, p) \xrightarrow{w_1} (q_F^J, p) \xrightarrow{b_1} (p, \varepsilon),$$

with q_0^J and q_F^J respectively the initial and a final state of some procedural automaton $\mathcal{A}^J$ with $f(J) = a_1$.

Looking at the above stacked run, we can see that $b_1 = \overline{a_1}$ (or it wouldn't respect semantics of VSPA, see Definition 2.4) and that $w_1 \in L(\mathcal{A}^J)$. By induction, $w_1 \in WM(\Sigma)$.

Because w_1 is well-matched, we see that $\beta(u_0 a_1 w_1 b_1 u_1 a_2) = 1$. Therefore, the depth of w_2 is less than the depth of w. With the same reasoning as before, we prove that $w_2 \in WM(\Sigma)$ and that $b_2 = \overline{a_2}$. This is also true for all w_i and for all b_i.

Finally, the word $w = u_0 a_1 w_1 \overline{a_1} u_1 ... u_{m-1} a_m w_m \overline{a_m} u_m$, with $u_i \in \Sigma_{int}^*$ and $w_i \in WM(\Sigma)$, is well-matched by Definition 2.7.

Proof of Property (2.6)

$$\begin{aligned} &\text{Let } w \in \Sigma_{int}^*, \ f(J) = a : \\ &\quad w \in \widetilde{L}(\mathcal{A}^J) \iff a w \overline{a} \in L(\mathcal{A}^J). \end{aligned}$$

This property is simple to prove. Since $w \in \Sigma_{int}^*$, the VSPA behaves like an NFA, and it's trivial that the property is true. Let's formally prove it.

$w \in \widetilde{L}(\mathcal{A}^J) \Rightarrow a w \overline{a} \in L(\mathcal{A}^J)$:

Since $w \in \widetilde{L}(\mathcal{A}^J)$, by Definition 1.6, if we write $w = a_1 a_2 ... a_m$, it exists a run :

$$q_0^J \xrightarrow{a_1} q_1^J \xrightarrow{a_2} ... \xrightarrow{a_m} q_F^J,$$

with q_0^J and q_F^J respectively the initial and a final state of the procedural automaton $\mathcal{A}^J$, and $q_i \in Q^J$ for all $i = 1, ..., m-1$.

As this run exists and $a_i \in \Sigma_{int}$ for all i, the VSPA behaves like an NFA when it reads a symbol a_i, with no stack modification. Therefore, it exists the stacked run :

$$(q_0^J, \varepsilon) \xrightarrow{a_1} (q_1^J, \varepsilon) \xrightarrow{a_2} ... \xrightarrow{a_m} (q_F^J, \varepsilon),$$

$$(q_0^J, \varepsilon) \xrightarrow{w} (q_F^J, \varepsilon).$$

Since $f(J) = a$ and this stacked run exists, by Definition 2.6 :

$$aw\overline{a} \in L(\mathcal{A}^J).$$

The same reasoning proves that $w \in \widetilde{L}(\mathcal{A}^J) \Leftarrow aw\overline{a} \in L(\mathcal{A}^J)$.

Proof of Property (2.7)

$$\begin{gathered}\text{let } u_i \in \Sigma_{int}^*, K_i \in \Sigma_{proc} \\ \text{s.t. } u_0K_1u_1....u_{m-1}K_mu_m \in \widetilde{L}(\mathcal{A}^J) \Rightarrow \\ \forall w_i \in L(\mathcal{A}^{K_i}) : f(J)u_0w_1u_1...u_{m-1}w_mu_m\overline{f(J)} \in L(\mathcal{A}^J).\end{gathered}$$

Since $u_0K_1u_1....u_{m-1}K_mu_m \in \widetilde{L}(\mathcal{A}^J)$, by Definition 1.6, it exists an accepting run in the NFA $\mathcal{A}^J$ reading this word :

$$q_0^J \xrightarrow{u_0} q_1^J \xrightarrow{K_1} p_1^J \xrightarrow{u_1} q_2^J \xrightarrow{K_2} ... \xrightarrow{K_{m-1}} p_{m-1}^J \xrightarrow{u_{m-1}} q_m^J \xrightarrow{K_1} p_m^J \xrightarrow{u_m} q_F^J,$$

with q_0^J and q_F^J respectively the initial state and a final state of $\mathcal{A}^J$, and $q_i^J, q_i^J \in Q^J$ for all $i = 1,..,m$.

Let's look at the stacked run $u_0w_1u_1...u_{m-1}w_mu_m$ starting from the configuration (q_0^J, ε). First, we read the word $u_0 \in \Sigma_{int}^*$. Since this word is over the internal alphabet, there are no stack modification in the stacked run, and the VSPA behaves like an NFA. Therefore, it exists the stacked run :

$$(q_0^J, \varepsilon) \xrightarrow{u_0} (q_1^J, \varepsilon).$$

Then, we read the word $w_1 \in L(\mathcal{A}^{K_1})$. By Definition 2.6, we can write $w_1 = a_1w_1'\overline{a_1}$, with $a_1 = f(K_1)$, and there exists a stacked run of w_1' from the initial state to a final state of $\mathcal{A}^{K_1}$. Following the semantics of VSPA (see Definition 2.4), from the configuration (q_1^J, ε), we can read the symbol $a_1 = f(K_1)$ and go to the configuration $(q_0^{K_1}, p_1^J)$. As it exists an accepting stacked run of w_1' in $\mathcal{A}^{K_1}$, reading it can bring us to the configuration $(q_F^{K_1}, p_1^J)$, with $q_F^{K_1}$ a final state of $\mathcal{A}^{K_1}$. Finally, we read the symbol $\overline{a_1} = \overline{f(K_1)}$ and we go to the configuration (p_1^J, ε). In summary, there exists the following stacked run of w_1 from the configuration (q_1^J, ε) :

$$(q_1^J, \varepsilon) \xrightarrow{a_1} (q_0^{K_1}, p_1^J) \xrightarrow{w_1'} (q_F^{K_1}, p_1^J) \xrightarrow{\overline{a_1}} (p_1^J, \varepsilon).$$

We then read the word $u_2 \in \Sigma_{int}^*$ to reach the configuration (q_2^J, ε). After that, we read w_2, which is similar to w_1, and we reach configuration (p_2^J, ε). We do this for all u_i and for all w_i to obtain the stacked run :

$$(q_0^J, \varepsilon) \xrightarrow{u_0} (q_1^J, \varepsilon) \xrightarrow{w_1} (p_1^J, \varepsilon) \xrightarrow{u_2} (q_2^J, \varepsilon) \xrightarrow{w_2} (p_2^J, \varepsilon) \xrightarrow{u_3} ... \xrightarrow{w_m} (p_m^J, \varepsilon) \xrightarrow{u_m} (q_F^J, \varepsilon).$$

Therefore, there exists a stacked run :

$$(q_0^J, \varepsilon) \xrightarrow{u_0w_1u_1...u_{m-1}w_mu_m} (q_F^J, \varepsilon)$$

As $f(J) = a$ and the above stacked run exists, by Definition 2.6 :

$$au_0w_1u_1...u_{m-1}w_mu_m\overline{a} \in L(\mathcal{A}^J).$$

Note that, with Property (2.7), we could think that the mutual statement is also true :

$$au_0w_1u_1...u_{m-1}w_mu_m\overline{a} \in L(\mathcal{A}^J) \Rightarrow u_0K_1u_1....u_{m-1}K_mu_m \in \widetilde{L}(\mathcal{A}^J).$$

However, this isn't true, since a word w_i could be in the language $L(\mathcal{A}^{K_i})$ and in the language of another procedural automaton $\mathcal{A}^{K'_i}$.

As a counter example, take the procedural automaton $\mathcal{A} = (\widehat{\Sigma}, \Gamma, \{\mathcal{A}^S, \mathcal{A}^R\}, f, S)$ described in Figure 4, with $\widehat{\Sigma}$ the VSPA alphabet (with $\Sigma_{int} = \{b\}$, $\Sigma_{call} = \{a\}$, $\Sigma_{ret} = \{z = \overline{a}\}$ and $\Sigma_{proc} = \{S, R\}$), and $f(S) = f(R) = a$.

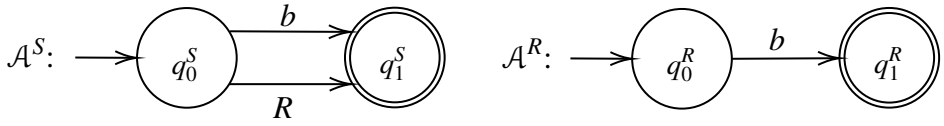


Figure 4: Example of VSPA that proves that the mutual statement of Property (2.7) is false.

If we take the word $w = aabzz$, it is clear that $w \in L(\mathcal{A}^S)$. This word can be written $w = au_0w_1u_1z$, with $w_1 = abz$ and $u_0 = u_1 = \varepsilon$. It is clear that $w_1 \in L(\mathcal{A}^S)$. However, $u_0Su_1 = S \notin \widetilde{L}(\mathcal{A}^S)$. Therefore, the mutual statement of Property (2.7) is false.

Proof of Property (2.8)

$$\begin{gathered}\text{let } w \in WM(\Sigma) \text{ s.t. } \exists (q,\gamma) \xrightarrow{aw\overline{a}} (p,\gamma) \Rightarrow \\ \exists J \in \Sigma_p \text{ s.t. } aw\overline{a} \in L(\mathcal{A}^J) \text{ and } p \in \delta(q,J).\end{gathered}$$

By following the semantics of VSPA (see Definition 2.4), we can decompose the stacked run in the property by :

$$(q,\gamma) \xrightarrow{a} (q_0, p\cdot\gamma) \xrightarrow{w} (q_F, p\cdot\gamma) \xrightarrow{\overline{a}} (p,\gamma),$$

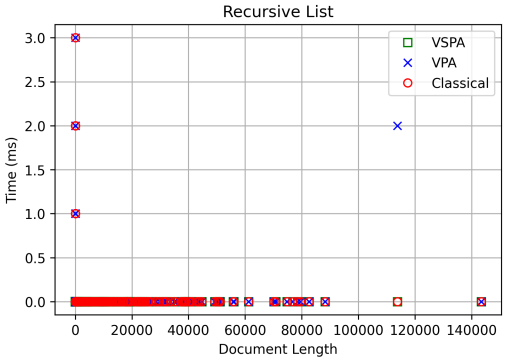
with q_0 and q_F respectively a initial and a final state of some procedural automata.

By following the semantics of VSPA (see Definition 2.4, more precisely, the call and return rules in Figure 2.3), since $w \in WM(\Sigma)$, q_0 and q_F should belong to the same procedural automaton $\mathcal{A}^J$, with $f(J) = a$. As w has an accepting stacked run from q_0 to q_F, $w \in L(\mathcal{A}^J)$.

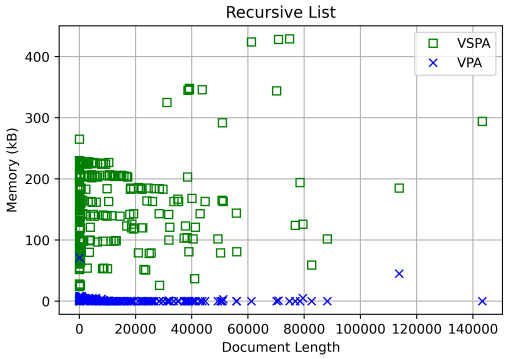
Furthermore, as p has been stacked when we read the call symbol a, by Definition 2.4, it exists the transition $q \xrightarrow{J} p$.

Appendix E

Results of *Recursive List* and *Basic Types*

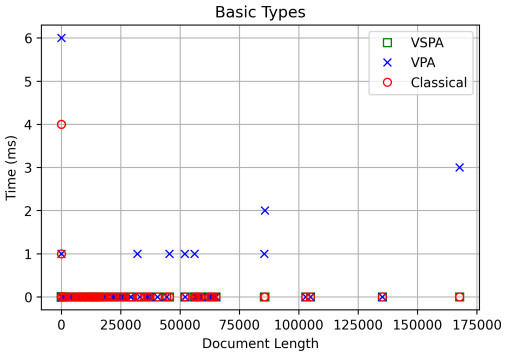


(a) Computation time depending on the number of symbols of the document.

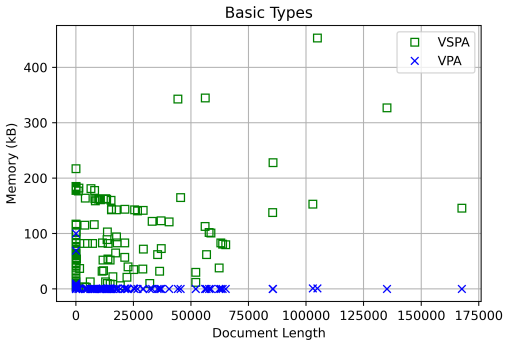


(b) Memory usage depending on the number of symbols of the document.

Figure 5: Time and Memory Results of validation of JSON documents of *Recursive List* schema. Green square correspond to the VSPA algorithm, blue crosses to the VPA algorithm, and red circles to the classical algorithm.



(a) Computation time depending on the number of symbols of the document.



(b) Memory usage depending on the number of symbols of the document.

Figure 6: Time and Memory Results of validation of JSON documents of *Basic Types* schema. Green square correspond to the VSPA algorithm, blue crosses to the VPA algorithm, and red circles to the classical algorithm.