UMONS
Faculty of Sciences
Department of Computer Sciences
Software Engineering Lab

Resolving Inconsistencies in Model-Driven Engineering using Automated Planning

Jorge Pinna Puissant

A dissertation submitted in fulfillment of the requirements of
the degree of Doctor in Sciences

Advisor
Prof. Dr. Tom Mens

Jury
Prof. Dr. Jef Wijsen
Prof. Dr. Hadrien Mélot
Dr. Ragnhild Van Der Straeten
Prof. Dr. Xavier Blanc

September 2012

To my parents,
the lighthouse
that guided my life.

Acknowledgment

Firstly, I would like to thank Tom Mens, for allowing me to do this PhD, by being the best advisor I could have and for proof-reading this dissertation. I also would like to thank him for his endless patience, his friendship and his support all along this difficult path.

I would like to thank my Jury members for taking the time to read and for giving feedback on this dissertation. Especially I would like to thank Ragnhild Van Der Straeten who helped and supported me since the beginning. She also proof-read this dissertation and provided me with very useful comments.

I would also like to thank all my colleagues and former colleagues from the computer science department. Especially Michaël, Mathieu, Sylvain, Romuald and Javier for having put a good atmosphere in the lab, for having had interesting brainstormings about my thesis and for having had good discussions about anything and everything.

I would never have been here without my parents and my family. I could never thank enough my parents for giving me the opportunity to study and support me in every decision I made. I thank my friends Jonathan, Yoan, Ulrik, Cat, Julien, Francois, Alex and Lam for having been there to support me, motivate me, corrupt me, encourage me to finish this dissertation and also proof-read parts of this dissertation. Thank you guys, you are like family for me. I hope that, from now on, I will have more time to spend with you.

Last but certainly not least, I would like to give a special thanks to Sandrine, for being there for me, helping me and supporting me. She has not only helped me proof-reading this dissertation. But also she made me better than I was. Sandrine, you will always be in my heart. I would never have been able to travel this path without you. Thank you.

This work has been supported by ARC research project AUWB-08/12-UMH "Model-Driven Software Evolution" financed by the Ministère de la Communauté française – Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgium.

Abstract

One of the main research challenges in model-driven software engineering is to deal with inconsistencies in software design models. Automated techniques to detect and resolve these inconsistencies are essential. A wide range of model inconsistency resolution approaches have been presented in literature.

In this dissertation, we study a novel approach that uses automated planning, an artificial intelligence technique, for the purpose of automatically generating resolution plans for model inconsistencies. We present two different planning approaches to generate resolution plans: *Fast-Forward Planning System* (FF), an existing domain-independent heuristic state-space progression planner; and *Badger*, a new domain-specific regression planner that we have implemented in Prolog. We study their feasibility in the domain of model inconsistency resolution. *Badger* has demonstrated good performance for inconsistency resolution, is metamodel-independent and can generate multiple resolutions plans. In addition, the planner algorithm of Badger can be adapted by taking full advantage of the domain knowledge.

We validate Badger on a large number of automatically generated UML class diagram models of various sizes, as well as on UML models obtained by reverse engineering five Java programs, and on a classical toy example. We use a set of 13 structural inconsistency types based on OCL constraints found in the UML metamodel specification. We analyse the scalability results of the approach obtained through several stress-tests and discuss the limitations of our approach. Our empirical analysis reveals a strong linear relationship between the model size and the execution time, a quadratic relationship between the number of inconsistencies and the execution time and a quadratic relationship between the number of generated plans and the execution time. We also observe an increase of the execution time as the number of actions in the resolution plans increases. In addition, our approach scales up to models containing more than 10000 model elements. We validate the metamodel independence of Badger by applying it to the problem of resolving code smells in Java programs.

We explore how Badger can adapt the way it presents the resolution plans to the users by modifying the evaluation function of the planner algorithm. We analyse and discuss the solutions and possibilities that allows our planner to change the order in which the resolution plans are generated.

We conclude that it is feasible to use automated planning in a scalable way for design-model inconsistency resolution.

Résumé

Un des principaux défis de la recherche dans l'ingénierie logicielle dirigée par les modèles est de faire face à des incohérences dans des modèles logiciels. Des techniques automatisées pour détecter et résoudre ces incohérences sont essentielles. Un large éventail d'approches de résolution d'incohérences de modèles a été présenté dans la littérature.

Dans cette thèse, nous présentons une nouvelle approche qui utilise la planification automatisée, une technique d'intelligence artificielle visant à générer automatiquement des plans pour résoudre des incohérences dans les modèles. Nous évaluons deux approches différentes de planification dans le but de générer des plans de résolution. La première approche est *Fast-Forward Planning System* (FF), un planificateur existant indépendant du domaine qui se base sur une heuristique pour faire une recherche en avant dans un espace d'état. La seconde approche est *Badger*, un nouveau planificateur spécifique au domaine que nous avons développé en Prolog et qui effectue une recherche en arrière. Nous réalisons une étude de faisabilité de ces deux planificateurs dans le domaine de la résolution d'incohérences de modèles. *Badger* est performant lors de la résolution d'incohérences, est indépendant du métamodèle et peut générer de multiples plans de résolution. De plus, l'algorithme de planification de Badger peut être adapté afin de profiter pleinement des spécificités du domaine.

Nous validons Badger sur un nombre important de diagrammes de classe UML de tailles diverses générés automatiquement, sur des modèles UML obtenus par rétro-ingénierie de cinq programmes Java, ainsi qu'un cas d'école classique. Nous utilisons un ensemble de types d'incohérences structurelles. Ces types sont basés sur des contraintes OCL issues de la spécification du métamodèle UML. Nous analysons le passage à l'échelle de l'approche et discutons de ses limitations. Notre analyse empirique révèle une forte relation linéaire entre la taille du modèle et le temps d'exécution ainsi qu'une relation quadratique entre le nombre d'incohérences et le nombre de plans générés d'une part et le temps d'exécution d'autre part. Nous observons également une augmentation du temps d'exécution lorsque le nombre d'actions dans les plans de résolution augmente. De plus, notre approche a été testée avec succès sur des modèles contenant plus de 10.000 éléments. Nous validons l'indépendance de Badger vis-à-vis du métamodèle en l'appliquant à la résolution de problèmes de conception (code smells) dans des programmes Java.

Nous explorons de quelle manière Badger peut être adapté par l'utilisateur dans la façon dont il lui présente les plans de résolution. Cette adaptation est réalisée en modifiant la fonction d'évaluation de l'algorithme du planificateur. Plus spécifiquement, nous analysons et discutons les solutions et possibilités qui permettent à notre planificateur de changer l'ordre dans lequel les plans de résolution sont générés.

Pour conclure, nous pouvons dire que l'utilisation de la planification automatisée est possible et passe à l'échelle dans le cadre de la résolution des incohérences dans des modèles logiciels.

Table of Contents

List of Figures

List of Tables

Introduction

As a consequence of the increasing adoption of model-driven software engineering, large-scale industrial projects make use of multiple models, being developed by hundreds of developers [103, 134]. In such a context, inconsistencies invariably arise in models and may be the cause of project failure [60]. Developing techniques for dealing with model inconsistencies then becomes crucial.

A model is considered to be inconsistent if it contains undesirable patterns, which are specified by so-called inconsistency rules [4]. These patterns and rules can reveal and capture problems of lexical, structural, behavioural or visual nature. Inconsistency detection consists in identifying the presence of these undesirable patterns in the model.

As defined by Spanoudakis and Zisman [140], inconsistency management not only consists in the detection of inconsistencies but also in their handling. Once inconsistencies have been detected in models, they have to be dealt with, either by resolving them, or by ignoring or postponing them to later [4]. *Inconsistency resolution* consists in automating the modification of a model in order to make it consistent.

If one desires to resolve model-driven inconsistencies with automated techniques, the approaches need to generate possible resolutions without the need of manually writing resolution rules or writing any procedures that generate possible resolutions. The approaches need to enable the resolution of multiple inconsistencies at once and to perform the resolution in a reasonable time. In addition, the approaches need to be generic, *i.e.*, it should be easy to apply them to different modelling languages.

In this dissertation, we propose to use the technique of *Automated Planning* from Artificial Intelligence domain to address this problem. Automated planning aims to automatically generate plans, *i.e.*, sequences of actions that lead from an initial state to a state meeting a specific predefined goal. Automated planning will be used for the purpose of automatically generating resolution plans for model inconsistencies. It seems that automated planning fulfills the conditions to address the problem of inconsistency resolution in design models. Furthermore, the use of automated planning for the purpose of model inconsistency resolution is a novel approach, we are not aware of any other work having used this technique for this purpose.

Therefore, in this dissertation we will investigate the following thesis statement:

> *Automated Planning can be used to resolve, in a scalable way, design-model inconsistencies.*

The remainder of this dissertation is structured as follows.

Chapter 1 presents the context of the problem that motivates this dissertation. It presents the software development methodology of *model-driven software engineering* and its research challenges. *Model inconsistency management* is identified as one of these challenges. The activities of model inconsistency management are described and a state

of the art of the techniques and formalisms used in model inconsistency management is presented.

Chapter 2 proposes a feature-based analysis of design model inconsistency resolution approaches based on three main criteria: flexibility, usability and extensibility. This study will allow us to identify weaknesses in eight recent approaches. We will take into account these weaknesses to avoid them in the development of our own solution.

Chapter 3 introduces automated planning, and presents and formally defines classical planning. Classical planning is composed of a representation language and an algorithm. We introduce the most common representation languages and algorithms and conclude this chapter by presenting some classical planning implementations.

Chapter 4 explains how automated planning can be used for resolving model inconsistencies. We present two different planning implementations: *Fast-Forward Planning System* (FF), an existing domain-independent heuristic state-space progression planner; and *Badger*, a new domain-specific regression planner that we have implemented in Prolog. We study their feasibility in the domain of model inconsistency resolution and make a small scalability study. We conclude this chapter with a discussion about the strengths and limitations of these implementations. Chapter 5 presents the improvements that we have made to Badger to address the limitations found in the previous chapter.

Chapter 6 assesses the scalability of Badger. For this purpose we use automatically generated UML models, reverse engineering models and a toy example model. We gradually increase during our experiments the size of the model, the number of inconsistencies to resolve, and the number of generated plans. We conclude this chapter with a summary of the scalability analysis results.

Chapter 7 analyses and discusses the solutions and possibilities we have explored to allow Badger to adapt the way it presents the resolution plans to the users, by changing the order in which the resolution plans are generated.

Chapter 8 summarises the outcomes of this dissertation, discusses its results, contributions and limitations and presents the open research perspectives.

1

Model Inconsistency Management

This chapter presents the software development methodology of *model-driven software engineering*, it also presents the research challenges of this methodology. *Model inconsistency management* is identified as one of these challenges. We describe the activities of model inconsistency management. We conclude this chapter with a state of the art of the techniques and formalisms used in model inconsistency management. The work presented in this chapter was previously published in:

(i) *"Challenges in model-driven software evolution"* in the 7th BElgian-NEtherlands software eVOLution workshop (BENEVOL) 2008 [64], co-authored with Michaël Hoste (University of Mons, Belgium) and Tom Mens (University of Mons, Belgium).

(ii) *"Amélioration de la qualité de modèles: Une étude de deux approches complémentaires"* in Technique et Science Informatiques, 2010 [96], co-authored with Tom Mens (University of Mons, Belgium); Dalila Tamzalit (University of Nantes, France) and Michaël Hoste (University of Mons, Belgium)

1.1 Model-Driven Software Engineering

In software engineering, there is a growing interest in the use of models as primary artifacts in the development of software systems. This has created a new software development methodology known as *model-driven software engineering (MDE)* [40, 77, 133, 142]. MDE promises to cope with the intrinsic complexity of software-intensive systems by raising the level of abstraction, and by hiding the accidental complexity of the underlying technology as much as possible [16]. This opens up new possibilities for creating, analysing, manipulating and formally reasoning about systems at a high level of abstraction. Model transformation techniques and languages [26] enable a wide range of different automated activities such as translation of models (expressed in different modelling languages), generating code from models, model refinement, model synthesis or model extraction, model restructuring, *etc.*

1.1.1 Model Evolution

It is generally acknowledged that software employed in a real-world environment must be continuously evolved and adapted, otherwise it is doomed to become obsolete due to changes in the operational environment or user requirements [81, 118]. On the other hand, any software system needs to satisfy certain well-defined quality criteria related to performance, correctness, security, safety, reliability and soundness and completeness w.r.t. the problem specification. It is a very challenging task to reconcile these conflicting concerns, in order to develop software that is easy to maintain and evolve, yet continues to satisfy all required quality characteristics. When we look at contemporary support for software evolution at the level of models, however, research results and automated tools for this activity are still in their infancy [155].

1.1.2 Challenges in Model-Driven Software Engineering

In Hoste *et al.* [64], we identified a number of fundamental research challenges in MDE. It is important to note that this list of challenges is inevitably incomplete. Also, the order in which we present the challenges here is of no particular importance.

Model quality. Model quality is important because models are fundamental artifacts in software development process, as any other software artifact the individual quality affects the internal and external software product. There is not a consensus on how to define model quality and how to relate model quality metrics to software quality factors. It is common sense that model quality has a great important in software product [41]. A model can have many different non-functional properties or quality characteristics that may be desirable (e.g., usability, readability, performance and adaptability). It remains an open challenge to identify which qualities are necessary and sufficient for which type of stakeholder, as well as how to specify these qualities formally, and how to relate them to one another.

The next logical question concerns how we can objectively measure, predict and control the quality of models during their evolution. One possible solution is by resorting to *model metrics*, the model-level equivalent of software metrics. The challenge here is to define model metrics in such a way that they correlate well with external model quality characteristics.

Model improvement. The technique of *model refactoring*, the model-level equivalent of program refactoring, is used in order to improve model quality. An important point of attention is the study of the relation between model metrics and model refactoring. In particular, the assessment to which extent model refactoring affects metric values. A formal specification of model refactoring is required to address these issues.

In a similar vein, a precise understanding of the relation between model smells and model refactoring is required, in order to be able to suggest, for any given model smell, appropriate model refactorings that can remove this smell. The other way around, the assurance that model refactorings effectively reduce the number of smells is needed. This challenge is currently addressed by several authors [93–95, 102, 126, 152, 153, 166].

Model inconsistency management. In MDE, model inconsistencies invariably arise, because a (software) system description is composed of a wide variety of diverse models; some of which are maintained in parallel, and most of which are subject to continuous evolution. Therefore, there is a need to formally define the various types of model inconsistencies in a uniform framework, and to resort to formally founded techniques and strategies to detect and resolve these model inconsistencies. A prerequisite for doing so is to provide traceability mechanisms, by making explicit the dependencies between models.

Various research groups are currently working to address this challenge [2, 10, 32, 33, 35, 37, 70, 83, 97–99, 109, 110, 128, 135, 154, 156, 157, 161, 164].

Language evolution. Not only models evolve, but so do the modeling languages in which the models are expressed, though at a lower pace. In order to ensure that models do not become obsolete because their languages have evolved, we need mechanisms to support the co-evolution between both. In a similar vein, the model transformation languages may evolve in parallel with the model transformations being used, so we also need to support co-evolution at this level [39, 100, 158].

Conflict analysis. Another important challenge has to do with the ability to cope with conflicting goals. During model evolution, trade-offs need to be made all the time:

- When trying to improve model quality, different quality goals may be in contradiction with each other. For example, optimising the understandability of a model may go at the expense of its maintainability.
- In the context of inconsistency management, inconsistency resolution strategies may be in mutual conflict.
- In the context of model refactoring, a given model smell may be resolved in various ways, by applying different model refactorings. Vice versa, a given model refactoring may simultaneously remove multiple model smells, but may also introduce new smells.

It should be clear from the discussion above that uniform formal support for analysing and resolving conflicts during model transformation is needed. Mens *et al.* [95,97,98] have started to explore formal techniques based on critical pair analysis and sequential dependency analysis to detect and reconcile conflicting concerns.

Collaborative modelling. Another important challenge in model-driven software evolution is to cope with models that evolve in a distributed collaborative setting? This naturally leads to a whole range of problems that need to be addressed, such as the need for model differencing, model versioning, model merging or model integration, model synchronisation, and so on. This challenge is already under active study by Blanc *et al.* [105, 141].

To address the previously described MDE challenges, tool support needs to be developed. But these tools encounter more technical challenges that also need to be tackled:

Model independence. How can we represent and manipulate different types of models in a uniform way, without needing to change the infrastructure (tools, mechanisms and formalisms) for reasoning about them? Such model independence is of scientific as well as practical importance, because we want the solutions to be sufficiently generic, in order to be applicable beyond more software models. Indeed, we want to be able to support an as wide range of models as possible, including process models, workflow models, ontology models, and many more. *Zhang et al.* [166] have illustrated the feasibility of achieving such model independence by implementing a generic model transformation that can be used to transform domain-specific models. Blanc *et al.* [10] also illustrated this by presenting an unique way to represents models and model changes as sequences of elementary model operations. *Eclipse modeling framework* (EMF)[1] is a framework for building and using modeling languages based on a structural data model. EMF is based on the metamodel Ecore for describing models. It's another example of model independence. Some examples of tools developed using EMF and Ecore metamodel are : ATL[2] which is a model transformation language; Kermeta[3] which is a model transformation language and model checking tool; Epsilon[4] which is a code generation, model-to-model transformation, model validation, comparison, migration, merging and refactoring tool.

Scalability and incrementality An important practical aspect in MDE is the ability to provide tool support that is scalable to large and complex models. This scalability is essential in order to allow, on a medium to long term, to transfer research results to industrial practice by integrating them into commercial modelling environments and by validating them on industrial models. Obviously, this requirement imposes important restrictions on the underlying formalisms to be used. Truly incremental techniques help to close the gap between formal techniques and pragmatic software development approaches, which are inherently evolutionary in nature. That it is indeed possible to come up with such an incremental approach, as illustrated by *Egyed* [32, 33], who proposes a lightweight incremental approach to model consistency checking that scales up to large industrial models. In a similar vein, *De Fombelle* [27] provides a formal treatment of incremental consistency checking of UML models. Blanc *et al.* [10, 11] also illustrated this by proposing an incremental inconsistency detection approach.

In this dissertation, we will limit ourselves to the study of the MDE challenge of *model inconsistency management*.

[1] `http://www.eclipse.org/emf/`
[2] `http://eclipse.org/atl/`
[3] `http://www.kermeta.org/`
[4] `http://eclipse.org/gmt/epsilon/`

1.2 Model Inconsistency Management

As aforementioned, an effect of the increasing adoption of model-driven software engineering is that large-scale industrial projects make use of multiple models, being developed by hundreds of developers [103,134]. In such a context, inconsistencies invariably arise in models and may be the cause of project failure [60]. Developing techniques for dealing with model inconsistencies becomes crucial.

Model inconsistency management is defined by Finkelstein *et al.* [45] as the process in which inconsistencies are handled. Initially, inconsistency management approaches were developed to address the problem of inconsistencies typically encountered during the merging of software models in distributed multi-user environments [44, 45]. These approaches have been generalized later to deal with inconsistencies in all contexts of model-driven software development [11, 35, 54, 97, 140].

A model is considered to be inconsistent if it contains undesirable patterns, which are specified by so-called *inconsistency rules* [4, 5]. These patterns and rules can reveal and capture problems of lexical, structural, behavioural or visual nature. Inconsistency detection consists in detecting the presence of these undesirable patterns in the model.

As defined by Spanoudakis and Zisman [140], inconsistency management not only consists in the detection of inconsistencies but also in their handling. Once inconsistencies have been detected in models, they have to be dealt with, either by *resolving* them, or by ignoring or postponing them to later [4, 5].

Inconsistency management becomes even more important when a model consists of several diagrams. During the evolution of the model, it is possible that these diagrams become incompatible with each other and that inconsistencies appear.

An alternative approach, more conservative than model inconsistency management, is to maintain the consistency of a model at all costs [38]. This *consistency maintenance* approach is not very appropriate in the MDE context [5, 112]. According to *Nuseibeh et al.* [112], maintaining consistency at all times is counterproductive. It constrains the developer and reduces significantly her freedom. It is also almost impossible to avoid the inconsistencies that arise in a metamodel evolution environment and in a collaborative development. For these reasons this approach will not be discussed in this dissertation.

1.3 Terminology and Example

Many definitions exist for the notion of *model* [1, 9, 22, 47, 77, 90, 115]. We adopt the one of Mellor *et al.* [90] that defines a model as "*a coherent set of formal elements describing something (*e.g.*, a system, bank, phone or a train) built for some purpose that is amenable to a particular form of analysis.*"

There are also many definitions for *model inconsistency* [4, 5, 32, 36, 44, 84, 98, 114, 140, 154]. Nuseibeh *et al.* [114] define a model inconsistency as "*any situation in which a set of descriptions does not obey some relationship that should hold between them. The relationship between descriptions can be expressed as a consistency rule against which the descriptions can be checked*". Different authors have used a different terminology to refer to consistency rules such as well-formedness rules [140], structural rules [154], detection rules [98], syntactic rules [36], and inconsistency detection rules [10]. In this dissertation we will use the term *inconsistency rule* to refer to a rule that allows to detect an inconsistency. We will also use the term *inconsistency type* to refer to the set of all

inconsistencies detected using the same inconsistency rule.

Based on the above definitions, we consider a model to be inconsistent if and only if there exists at least one inconsistency that is present in the model. When a model is inconsistent, the goal of any inconsistency resolution approach is to change the model in order to remove some of its inconsistencies. We acknowledge here the fact that, in some cases, some of the inconsistencies can be tolerated (*e.g.*, because they have a low priority, or because they cannot be resolved right now) and therefore ignored by the resolution [4,5].

To illustrate the terminology introduced above, we present a classical example of design model inconsistency resolution borrowed from Egyed *et al.* [35] and Xiong *et al.* [164]. It is a system that displays a stream (*e.g.*, a video stream) to a user. The UML model of this system is composed of a class diagram representing the structure (Figure 1.1), and a sequence diagram and state machine diagram representing the behaviour (Figures 1.2 and 1.3). The class diagram presents three classes (`User`, `Display` and `Streamer`). Class `User` is associated to class `Display`, which is associated to class `Streamer`.

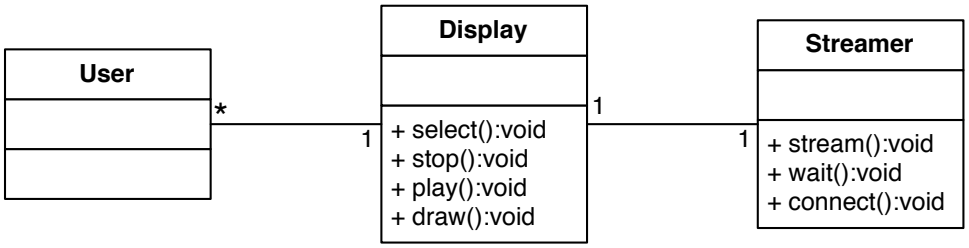


Figure 1.1 – Class diagram of the example

The sequence diagram (Figure 1.2) presents one scenario of interaction between the user, the display and the streamer. The protocol state machine diagram (Figure 1.3) defines the streamer protocol. "A protocol state machine is always defined in the context of a classifier. It specifies which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifiers operations." [117]

UML models are constrained by inconsistency rules [117, 139]. In this example, at least three inconsistencies, corresponding to different inconsistency rules (named R_1, R_2 and R_3), can be distinguished.

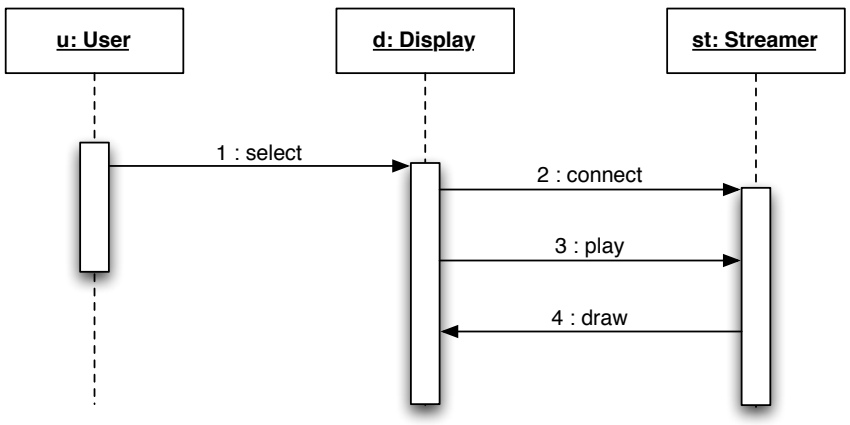


Figure 1.2 – Sequence diagram representing a specific scenario of user interaction.

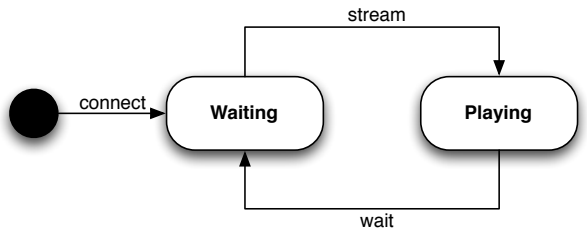


Figure 1.3 – Protocol state machine representing the protocol of class `Streamer`.

- **Rule** R_1**:** each message in a sequence diagram needs to have a corresponding operation that needs to be owned by the message receiver's class.
- **Rule** R_2**:** scenarios (*i.e.*, traces of messages sent to a particular object) expressed by a sequence diagram need to be included in state machine diagrams that may be attached to classes of objects to which the lifelines in the sequence diagram belong.
- **Rule** R_3**:** each message in a sequence diagram needs to have a corresponding navigable association in a class diagram.

The UML model m that is composed of Figures 1.1, 1.2 and 1.3 contains three inconsistencies regarding those three inconsistency rules. The first inconsistency I_1, corresponding to rule R_1, arises because message `play` used in the sequence diagram is not defined by an operation in the class `Streamer`. This could be resolved in numerous ways: by renaming message `play` into `stream`, by making the object `d:Display` the receiver of message `play`, by removing message `play`, and so on.

The second inconsistency I_2 of rule R_2 arises because the scenario presented in the sequence diagram is not included in the `Streamer`'s protocol state machine (the message `play` is not allowed to follow `connect`). This could be resolved by either adapting the sequence diagram or by adapting the state diagram.

The third inconsistency I_3 corresponding to rule R_3 arises because message `draw` in the sequence diagram has no corresponding navigable association in the class diagram. To resolve this inconsistency it suffices to make the association navigable in both directions.

1.4 Categories of Model Inconsistencies

To deal with model inconsistencies, the aim of this dissertation, we first need to know the different inconsistency kinds that can arise on the different dimensions of modelling.

A real software system is often too complex to be described in a single representation. A given system is specified by a set of different artifacts (*e.g.*, languages, programs, diagrams) and different views [54, 76]. In MDE, a system is usually represented by one or more models that are described in one or more modeling languages. Typically, in the context of UML [117], a system model is described by the diagrams (*e.g.*, class, sequence, state-transition) that can be considered as models representing a specific point of view on the system.

As aforementioned, when a model evolves, inconsistencies may arise that are sometimes necessary temporarily [4, 5], especially in iterative and evolutionary process in multi-user environments. If a model contains at least one inconsistency it is considered as an *inconsistent model*, otherwise it is considered as a *valid model*.

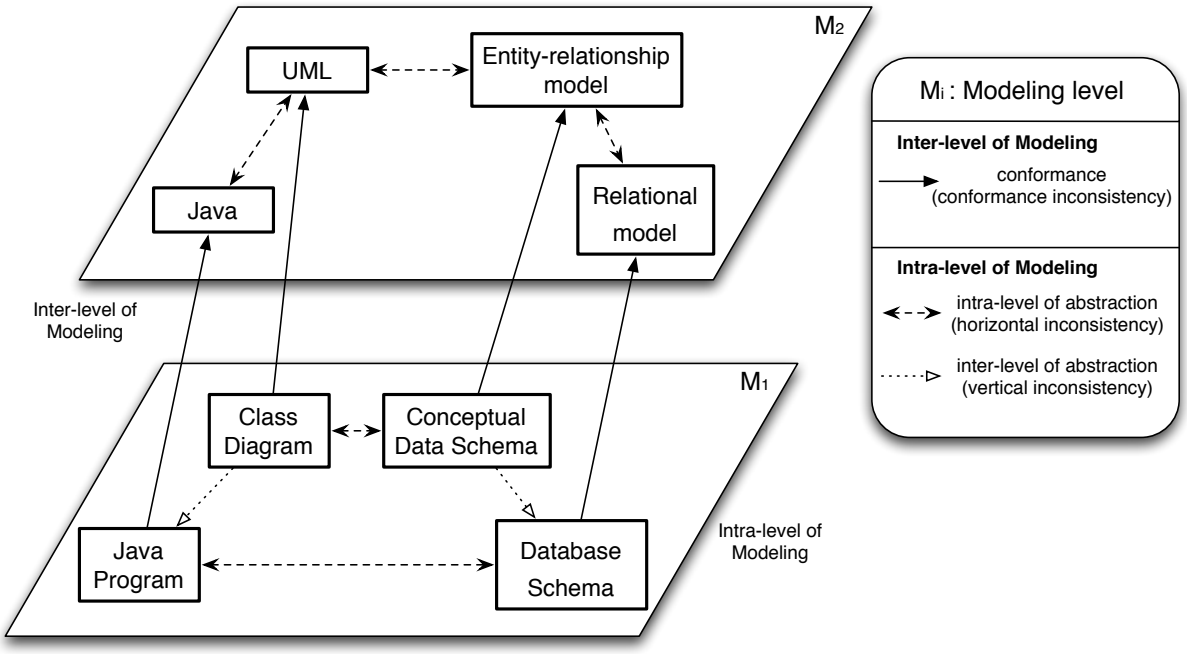


Figure 1.4 – Dimensions and model inconsistencies

Let us define and identify the main categories of inconsistencies in a model.

Modeling Level. As defined by the OMG [116], modeling levels represent the conformance relationship between a model and its metamodel(s). For example, in Figure 1.4, a class diagram is consistent with the UML metamodel, and a conceptual data schema [143] conforms to the entity-relationship model. There is a *conformance inconsistency* if a model does not respect the rules and constraints imposed by the metamodel.

Level of abstraction. Within the same level of modeling, there are several levels of abstraction. In the modeling level M_1 of figure 1.4, the class diagram and the conceptual data schema are at the same level (*intra-level*) of abstraction: they both represent a model in the design phase of a same system S. It is the same for the database schema and the Java program. The class diagram (respectively, the conceptual data schema) and the Java program (respectively, the database schema) are two different levels (*inter-level*) of abstraction but remain at the same level of modeling (M_1 level). We can distinguish two kinds of inconsistencies: *Horizontal* and *Vertical* inconsistency.

- **Horizontal inconsistency** (*intra-level of abstraction*). This kind of inconsistency occurs when two linked models in the same level of abstraction are incompatible. This is for example the case when a message in a sequence diagram does not have a corresponding operation owned by the message receiver's class (Rule R_1 of the example in the previously section).
- **Vertical inconsistency** (*inter-level of abstraction*). A refinement relation can exist between two related models that reside at different levels of abstraction. If the refinement between all or part of the two models is not checked, this is called *vertical inconsistency*. This is for example the case when an attribute

present in a class C of a class diagram is missing in the Java class refining the class C of the class diagram.

Syntax. Any model can be constrained by a *syntax*. There are two types of syntax: the *concrete* syntax and the *abstract* syntax.

- **Concrete syntax.** The concrete syntax represents the used visual notation and the special arrangement constraints of the modeling elements. A *concrete syntax inconsistency* can occur if these constraints are not met, for example, due to a superposition of elements or the use of a visual notation that is not recognized.
- **Abstract syntax.** The abstract syntax can be derived from the concrete syntax by a parser that generates a grammar (such as textual programming languages) or a metamodel (as for most visual modeling languages). The abstract syntax formalizes the conformance rules between a model and its metamodel: a model that does not conform to its metamodel yields an *abstract syntax inconsistency*. In the remainder of this dissertation we will refer to it as a *structural inconsistency*. However, the constraints imposed by a metamodel can be extended to encompass other ones (*e.g.*, business rules). As part of UML, this is possible through the mechanisms of profiles, stereotypes and OCL constraints.

Semantics. Beyond the syntactic level, a model may be valid according to semantic constraints. Such a constraint can affect all levels but also one or more models. Therefore, any model is constrained by semantics but can also be constrained according to some practices. *Semantics* is the meaning given to all or part of a model in a context [56]. Semantics focuses on the interpretation of a model while the syntax focuses on its representation. Meaningful semantics is an unambiguous interpretation of an expression built according to a given syntax. In the remainder of this dissertation we will refer to it as a *behavioural inconsistency*. An example of a syntactic symbol that has different unambiguous semantics and does not have any semantic inconsistency is the sign '+', which has adding as semantic consensus. Its semantics is different in the expression "$x + y$"according to the types of x and y (*e.g.*, numbers, strings, tables). An example of semantic inconsistency is *inheritance* because of its ambiguity. According to *Meyer* [101], it may have eleven different interpretations, such as *specialization* (for subtype inheritance) or *factorization* (for extension inheritance). B is a subtype of A if the set of all instances of B is a subset of the set of all instances of A, for example, tenured professor and associated professor are subtypes of professor. B is an extension of A if B introduces features (attributes and operations) not present in A, and those features are not applicable to the instances of A, for example, an interactive whiteboard is an extension of a whiteboard.

One or more models may have different types of inconsistencies, without any particular constraint between these types of inconsistencies. However, it is preferable to ensure that the model syntax is correct before attempting to rule on its semantics validity. Different models (considered as syntactically correct) may have the same the semantics: they can provide different representations with the same meaning. For example, a model with redundancies has same semantics after removal of these redundancies.

Syntactic inconsistencies are the most obvious to detect, especially when the metamodel has a formal representation. It becomes more difficult if part of the metamodel is semi-formal, under-specified, or informal, such as UML [117], especially when it induces semantic ambiguities. Semantic inconsistencies can be particularly difficult to detect, diagnose and treat. These difficulties may be even more drastic when a model evolves.

In this dissertation, we will limit ourselves to deal with conformance, horizontal and abstract syntax inconsistencies.

1.5 Inconsistencies Management Activities

Three inconsistency management activities are generally identified: detection, diagnosis and resolution of inconsistencies [45, 112].

Inconsistency detection. Inconsistency detection focuses on the definition of inconsistency rules, the detection of inconsistencies in the model using these rules, and the identification of the source and cause of these inconsistencies. An inconsistency was previously defined as the occurrence of an inconsistency rule in the model. The source of an inconsistency is the set of model elements that are involved in the inconsistency [112, 140]. The inconsistency detection policy defines when and what rules will be checked.

Inconsistency diagnosis. The diagnostic activity is focused on the analysis of previously detected inconsistencies to identify the severity, importance, risk and impact. The impact of an inconsistency is the consequence of the inconsistency on all models. Based on this information, the user determines in which order the inconsistencies must be addressed, and if an inconsistency is treated individually or treated in group with all the dependent inconsistencies. The manager may also decide not to consider inconsistencies, or to process the inconsistencies late.

Inconsistency resolution[5]**.** A *resolution* is defined as a model change that removes some inconsistencies of the input model. A model inconsistency *resolution approach* produces one or more resolutions in order to remove inconsistencies from an inconsistent model. The inconsistency resolution activity is divided into the following steps [140]:

1. Compute the possible *resolutions* to remove the selected inconsistencies in the diagnosis activity.
2. Perform a *cost-benefit analysis* of the application of each of these resolutions. This includes the possible introduction of new inconsistencies by this resolution, the model distance between the input model and the solution model (which and how much elements were added, modified or deleted?) and the cost of model changes of the resolutions (each model change can have a different cost, for example, the cost of changing a model element may be smaller than the cost of deleting a model element).
3. Selection and application of the resolution model changes, based on the previous choices.

[5]Some authors use the term *resolution* [98], while others authors use the term *fixing*, *repair* [35] or *handling* [44]

An inconsistency management *policy* defines which activities will be implemented and their scheduling. It responds to the following questions: (a) When and how often should inconsistencies be detected, diagnosed or resolved? (b) Who is responsible for what activity? (c) What techniques will be used for the detection, diagnosis and resolution of inconsistencies? (d) What inconsistency rules will be checked during the detection of inconsistencies? (e) Which inconsistency should be considered to be resolved ?

In this dissertation we do not focus on the problem of inconsistency *detection*, due to the abundance of scientific research that is available in this domain [10, 32, 37, 83, 98, 157]. We prefer to focus our research on a problem that has been much less addressed, namely the *resolution* of inconsistencies.

1.6 Techniques and Formalisms

Some techniques to detect, diagnose and resolve inconsistencies are based on well-known mathematical domains such as formal logic and graph transformation. Other approaches are tailored for a precise objective or modeling tool. The most common techniques used in inconsistency management are:

Techniques based on graph transformation. Graphs are an obvious choice for representing models, since most types of models have a graph-based structure.

The technique based on graph transformation is based on rules [55]. These graph transformation rules are separated in two parts: preconditions and the change to apply. The preconditions are graph structures that have to be present (positive) or absent (negative) before applying the change.

Thanks to these graph transformation rules, inconsistencies can be detected automatically by looking for mattes of the left-hand side of the rule in the model. For that, detection rules are defined for each type of inconsistency. The application of each rule on a given model results in the detection of one or several inconsistencies (if they are present). The procedure is very similar for the inconsistency resolution. As for detection, one or several graph transformation rules have to be defined for the resolution of each inconsistency. The application of these rules corrects the inconsistencies in the model.

One of the main problems of this technique is that only the inconsistencies which can be expressed as a graph structure can be detected and resolved. This limits the application of the technique to the *structural* inconsistencies only.

Graph transformation has been used by several authors [59, 97, 98, 127] to deal with model inconsistencies. It also has been used to deal with inconsistencies in requirements engineering [52]. Mens *et al.* [97, 98] use the technique to present an iterative and interactive method of resolution of inconsistencies. The resolution of an inconsistency can have, as a consequence, the introduction of new inconsistencies which will be corrected, following the approach during the next iteration. In the process of resolution, the user has different possibilities. He has the responsibility to choose, in an interactive way, the resolution to apply. This approach uses *critical pair* analysis to determine the dependencies between the detection and the resolution of inconsistencies.

Techniques based on logic. The main idea in the techniques based on logic is to translate the models of a modeling language in a logic language, such as in a knowledge

base. The steps of detection, diagnostic and resolution of inconsistencies are realised by the formal inference of the logic rules.

As opposite to the previous approach, this one presents the advantage of not being limited to the treatment of structural inconsistencies. But it has a major inconvenient: the need to translate a model to the logic language and vice versa before and after the management of inconsistencies. This translation can introduce errors. Moreover, it is not always obvious to find back the source of the inconsistency.

Initially, logic has been used to manage the inconsistencies in requirement engineering [30, 44, 113]. Liu *et al.* [83] use logic to detect inconsistencies in UML models.

Van Der Straeten *et al.* [150, 151, 154] introduce a mechanism of inconsistency management based on the theory of *Description Logics* (DL) [3]. DLs are a *family of formalisms for representing knowledges*. They are *decidable* fragments of first order predicate logic. This *decidability* property is very interesting to reason formally on inconsistencies and to analyse their impact on the model.

Blanc *et al.* [10] present an incremental detection and an unique way to express the models in a logic language. This approach translates the model using a sequence of model construction operations. This sequence is composed of elementary operations such as create, delete, addProperty, remProperty, addReference, remReference. This allows to detect not only structural inconsistencies (in the model itself), but also *methodological* inconsistencies (*i.e.*, inconsistencies in the process followed to build the model).

Other techniques. Some techniques have been developed to be used directly with one or several specific modeling tools. Some other techniques have been developed for a very specific purpose (*e.g.*, deal with inconsistencies in a diagram or in a very specific situation). These techniques are typically *hardcoded*.

An example of a technique developed to be directly used in the context of modeling tool is the tool *UML/Analyzer* [32, 33]. It has been integrated in the modeling environment *IBM Rational Rose* to manage inconsistencies in UML diagrams. The tool has been used to detect inconsistencies instantaneously. The approach automatically searches inconsistencies in the model to each change done in it. According to the change, it determines which consistency constraint has to be checked.

Graaf and van Deursen *et al.* [53] propose to deal with the semantic inconsistencies using the model transformation language *ATL*[6]. Sabetzadeh *et al.* [130] propose another approach developed with a very specific purpose: inconsistency detection in models after fusion of different points of view of the same model. Other techniques and approaches developed directly in the context of a specific modeling tool are [78, 79].

1.7 Conclusion

The aim of this dissertation is to tackle the problem of inconsistency resolution by generating possible resolutions without the need of manually writing resolutions rules or writing any procedures that generate possible resolutions. The approach needs to enable the resolution of multiple inconsistencies at once and to perform the resolution in a reasonable

[6] `www.eclipse.org/m2m/atl`

time. In addition, the approach needs to be generic, *i.e.*, it needs to be easy to apply it to different modelling languages. In Van Der Straeten *et al.* [156] we explored the usage of model finders for this purpose. In this dissertation, we propose to use the logic reasoning base technique of *Automated Planning* from the domain of Artificial Intelligence.

2

A Feature-Based Analysis of Design Model Inconsistency Resolution Approaches

---

In this chapter, we propose a feature-based analysis of design model inconsistency resolution approaches based on three main criteria: flexibility, usability and extensibility. This study will allow us to identify weaknesses in eight recent approaches. We use it to study eight recent design model inconsistency resolution approaches. We will take into account these weaknesses to avoid them in the development of our own solution.

The work presented in Sections 2.1, 2.2 and 2.3 was made with the collaboration of: Tom Mens (University of Mons, Belgium); Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium); Marcos Aurélio Almeida da Silva (University Pierre et Marie Curie, France); Xavier Blanc (University of Bordeaux 1, France); and Jean-Rémy Falleri (University of Bordeaux 1, France) in the context of a Tournesol research project[1].

---

[1]Tournesol - Hubert Currien partnership co-financed by Wallonie - Bruxelles International, the Fonds de la Recherche Scientifique, the Ministère Français des Affaires étrangères et européennes, and the Ministère de l'Enseignement supérieur et de la Recherche, Belgium.

2.1 Introduction

As defined by [140], inconsistency management not only consists in the detection of inconsistencies but also in their handling. Once inconsistencies have been detected in models, they have to be dealt with, either by resolving them, or by ignoring or postponing them to later [4]. Many approaches focus on *inconsistency resolution*, which consists in automating the modification of a model in order to make it consistent. Some of these approaches aim at automatic generation of solutions thanks to logic rules [44,110,164]. Some are based on generator functions and aim at composing them in order to resolve inconsistencies [33,35]. Others are based on resolution rules that have been hand-coded [98,154].

Confronted with such a diverse set of approaches, it is difficult to measure the advantages and the drawbacks of each of them. Some approaches use different underlying mechanisms but seem to obtain similar results. Moreover, as some approaches claim to completely support the resolution of inconsistencies, this raises the questions of the existence of this research domain. Is it closed or is their still room for novel research? Finally, there are no guidelines to choose the most adequate approach for a researcher or practitioner that wishes to select an approach to deal with inconsistencies in his/her model.

This chapter aims at answering those questions by objectively studying the features of existing approaches that deal with model inconsistency resolution. In order to perform our analysis we select a representative set of approaches that correspond to the rigorous definition for model inconsistency resolution, presented in section 1.3. Even with the formal definitions presented in section 1.3, the scope is still too wide. Therefore, we will focus on software design models (e.g., UML models) because most recent inconsistency resolution approaches address such models. Other types of models, e.g., requirements models [159] and context models [165] will not be considered in this dissertation. Nevertheless the feature-based analysis that we propose should be applicable to those as well, as it does not depend on a particular type of model or modeling language.

2.2 Study Criteria

2.2.1 Selection of Criteria

In order to study inconsistency resolution approaches, many different criteria can be used. We present the most important ones below, and explain them in terms of the type of questions that need to be answered for an approach in order to fit the criteria.

- **Usability.** Is the resolution approach user-friendly?
- **Flexibility.** What is the functionality offered by the resolution approach? Does it provide only one resolution or multiple ones? Can it introduce new inconsistencies when resolving existing ones? Can it resolve multiple inconsistencies simultaneously?
- **Extensibility.** How easy is it to extend, customize or reuse the resolution approach?
- **Scalability.** Does the resolution approach scale up (in terms of execution time or memory usage) as the number of model elements, number of inconsistency rules or number of model inconsistencies grows in size?

- **Expressiveness.** How expressive is the language used to describe the inconsistency rules? Does it allow to specify all inconsistency rules the user would like to resolve?

- **Quality.** Is the resolution approach able to resolve model inconsistencies in the way that the user expects it to? Doesn't the approach propose resolutions that are useless or meaningless to the user? Does the approach always propose an acceptable resolution if one exists? Doesn't the approach propose too many resolutions, making it difficult or impossible for the user to select the most appropriate one?

Of these criteria, we have decided to rule out the latter three for our study. The reasons are given below.

In order to address the *scalability* criterion, we would need to define a benchmark that would allow us to carry out an experimental study of different approaches. Defining such a benchmark is a challenge of its own because the existing model inconsistency resolution approaches that can be found in research literature are quite diverse. They are implemented in different programming languages and on different operating systems, and use different input and output formats for the models, inconsistencies and resolutions. Even if it would be possible to come up with a commonly acceptable benchmark, it would only present a starting point for the real experimental scalability study. We therefore decided not to include the scalability criterion in this chapter as it would lead us too far.

We will not address the *expressiveness* criterion now because it is beyond our scope to study the expressiveness of languages, especially if these languages belong to different programming paradigms. In fact, expressiveness should be part of the aforementioned benchmark: the benchmark should impose a fixed set of inconsistency rules that need to be resolved by each resolution approach. If some approach is not able to do this, it has lesser expressiveness.

We will not address the *quality* criterion as it would require a controlled user study. Indeed, in presence of a wide variety of different resolution approaches, the only meaningful notion of quality would be the quality as perceived by the user. Since different users may have different ideas of what a good resolution strategy entails, an extensive and controlled user study would be needed. One needs to have access to all the tools to perform this controlled user study. Unfortunately, we do not have access to these tools and studying the quality of these approaches is not the aim of this dissertation.

As our study focused on model inconsistency resolution, we did not take into account information related to the inconsistency detection. While our study is well-suited to study inconsistency resolution approaches, other criteria would need to be taken into account to be able to study inconsistency detection approaches. An important feature to study detection approaches would be the *incrementality* of the detection. Rather than rechecking the detection rules on the entire model each time, incremental approaches will narrow down the focus on the relevant subset of the model and detection rules only.

2.2.2 Feature Modeling

To present the criteria for studying model inconsistency resolution approaches we resort to feature modeling [25]. Feature modeling are used to model the variability of applications "features" at a relative high level of granularity. Feature models are visually represented by means of feature diagrams. The feature diagram of Figure 2.1 presents our three selected study criteria: Flexibility, Usability and Extensibility. Each of these criteria play a crucial role in determining which inconsistency resolution approach is more appropriate,

from a specific point of view. The next subsections will further decompose each of the three criteria.

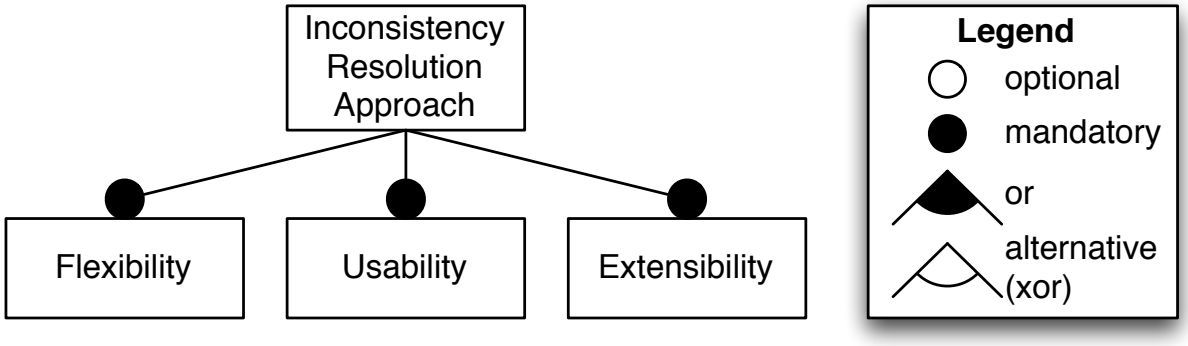


Figure 2.1 – Main criteria for classifying model inconsistency resolution approaches

2.2.3 Flexibility

To start with, we introduce three features to analysis the *flexibility* of a model inconsistency resolution approach. All considered features are summarised in Figure 2.2.

Feature 1 (Coverage)**.** *A resolution approach has* full *coverage, if it aims at proposing resolutions for* all *inconsistencies in the model at the same time. It has* multiple *coverage, if it aims at proposing resolutions for some inconsistencies at the same time. Finally, it has* single *coverage, if it aims at fixing only one inconsistency at a time, without taking into consideration if the proposed resolutions have an impact on other inconsistencies, e.g. if they create new inconsistencies.*

Feature 2 (Monotonicity)**.** *A resolution is* monotonic *if it does not introduce any new inconsistency (and does not reintroduce an existing inconsistency). A resolution approach can allow for non-monotonicity, or can impose monotonicity. It can even accommodate both options, by allowing for non-monotonicity in certain cases while imposing monotonicity in others.*

Feature 3 (Resolution space)**.** *As there may be more than one way to resolve a (set of) inconsistencies, a resolution approach may return either* multiple *resolutions, or only* one *resolution. In practice, a resolution approach is not able to return* all *possible resolutions if the resolution space is infinite.*

Feature 4 (Language independence)**.** *A resolution approach is language independent if it is not specific to a particular type of model or modeling language.*

2.2.4 Usability

To evaluate the *usability* of a model inconsistency resolution approach we distinguish three features, displayed in Figure 2.3, that are important for someone that wishes to use the approach in practice.

Feature 5 (Selection)**.** *A resolution approach can allow the user to select (respectively, ignore) either a subset of the inconsistencies that target the model, and/or a subset of the inconsistency rules he wishes to consider, and/or a sub-part of the model.*

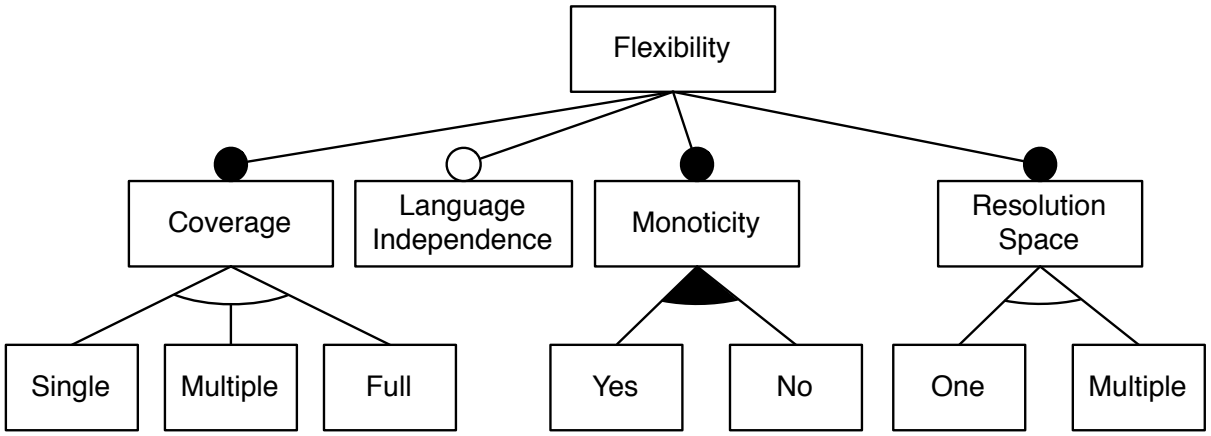


Figure 2.2 – Flexibility features

Feature 6 (Tool integration)**.** *A resolution approach can be integrated into a computer-aided software engineering (CASE) tool. Such integration, if present, allows the user to resolve model inconsistencies from within his CASE tool.*

Examples of CASE tools for UML modeling are Rational Software Architect, Poseidon, ArgoUML, Eclipse EMF, MagicDraw, VisualParadigm.

Feature 7 (User intervention)**.** *A resolution approach may allow the user to intervene while an approach produces resolutions, in order to guide the solver towards particular resolutions. A resolution approach may also proceed without allowing for user intervention. Approaches may accommodate both options, by allowing the user to choose whether she wants to intervene.*

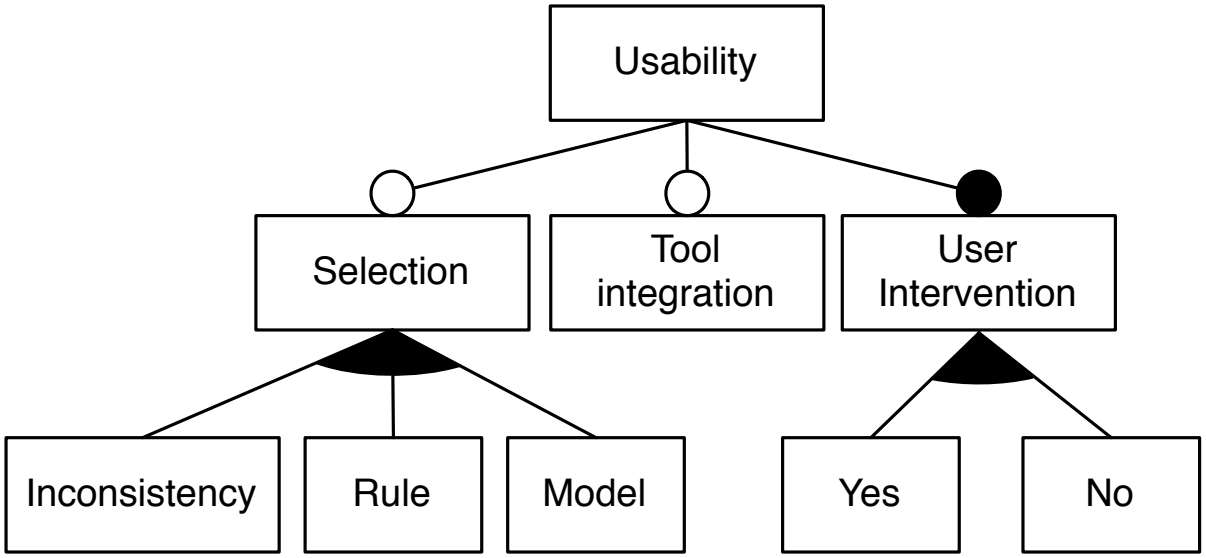


Figure 2.3 – Usability features

2.2.5 Extensibility

To assess the *extensibility* of a model inconsistency resolution approach we consider four features, displayed in Figure 2.4, that vary between the considered approaches.

Feature 8 (Solver)**.** *Some approaches may support multiple solvers to choose from depending on the user's requirements.*

Examples of *solvers* are planners, constraint solvers, SAT solvers, or rule engines.

Feature 9 (Resolution)**.** *An approach is* resolution extensible *if it can be extended to generate more resolutions. This means that the approach can extend its solution space, for example by adding more resolution rules.*

Feature 10 (Automation)**.** *An approach supports* automatic *inconsistency rules if it can generate resolutions to new inconsistency rules, without forcing the user to specify new resolutions for these inconsistencies. It supports* semi-automatic *inconsistency rules if the support to the automatic generation of resolutions for new inconsistencies is limited to some cases. Finally, it supports* manual *inconsistency rules if no automatic support is provided.*

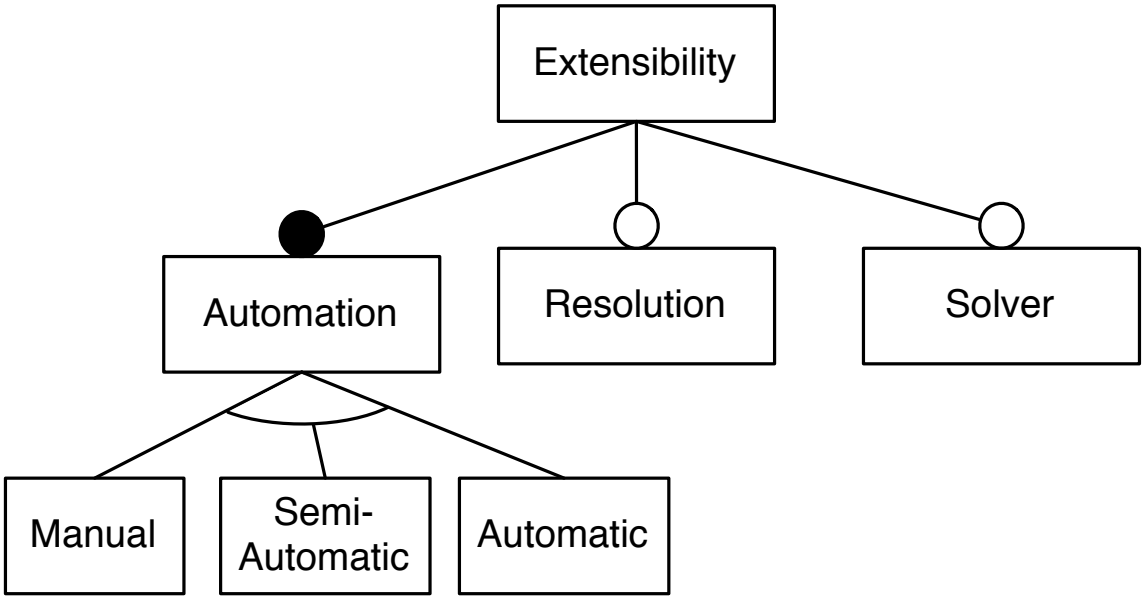


Figure 2.4 – Extensibility features

2.3 Approaches

The features defined in the previous section allow us to identify the variations across different model inconsistency resolution approaches. The criteria of section 2.2 are, in principle, applicable to any conflict resolution approach, regardless of whether the models represent requirements, design models, documentation, code or any other type of artefact. In order to restrict the scope of our study, however, we decided to consider only relatively recent (i.e., less than 10 years old) inconsistency resolution approaches that specifically focus on software design models, with an emphasis on UML models. In this category, we were able to find eight different approaches that have been proposed over the years by different researchers in the model-driven software engineering community. For ease of reference, we will refer to each approach using the name of its principal author: Almeida

da Silva [2], Egyed [33, 35, 128, 161], Kleiner [70], Mens [97, 98], Nentwich [109, 110], Van Der Straeten *I* [99, 135, 154], Van Der Straeten *II* [156], Xiong [164].

2.3.1 Almeida da Silva's Approach

a) Description of the approach

Almeida da Silva [2] proposes a search-based approach for handling inconsistencies. This approach takes as input a model and the last changes performed over it by the user. The approach uses sequences of atomic changes to represent both models and resolutions [10]. The relative order of the changes is used as an heuristic during the computation of resolutions.

The approach uses inconsistency rules that detect inconsistencies and specify the *causes* of the inconsistencies. *Generator functions* are then used to compute, for every change that has been spotted as a possible cause of inconsistency, a set of resolutions. The goal of the underlying algorithm is to construct resolutions that reduce the number of inconsistencies in the model. It does that by combining resolutions for n single causes into more complete resolutions that intend to fix all inconsistencies these causes generate. Note that a resolution for one inconsistency may cause new inconsistencies when combined with other resolutions.

b) Overall process

The overall process employed by this approach is described below.

Input: A model m with a set of inconsistencies I and a sequence of changes C.
Steps:

1. Approach uses the inconsistency detection rules to detect, in the change-based representation of m, the changes that cause the inconsistencies.
2. Approach uses pre-defined generator functions to compute a set of resolutions for each possible cause.
3. Approach uses the search algorithm to combine the resolutions computed in the previous step into a set of resolutions that reduce the number of inconsistencies in m.
4. User selects the resolution that will be applied.

Output: A model m' with a set of inconsistencies I' where $|I'| \leq |I|$.

c) Example

In our motivating example (*cf.* Section 1.3), the three inconsistency rules R_1, R_2 and R_3 have to specify the causes. Regarding R_1, one can specify, for example, that the cause of this kind of inconsistency lies in the last change that modifies the name of a message in the sequence diagram.

Now suppose that the current inconsistency has been added to the model by modifying the name of the second message from `stream` to `play`. This modification is represented

by a sequence of two changes. The first change removes `stream` as the name of that message and the second change sets it to `play`. The cause detection rule corresponding to R_1 detects that this new model is inconsistent. Moreover, it identifies the second change as the cause of this inconsistency.

This cause is then passed to the generator functions that compute the possible resolutions for this inconsistency. Three examples of proposed resolutions are: (i) deleting the `play` message, (ii) removing its name and adding one that matches the name of one of the operations in the class `Streamer`, and (ii) renaming some operation in the class `Display` to `play`.

The underlying search algorithm then tries each resolution in combination with the resolutions computed for the other inconsistencies in the model. This is done to determine the combinations that reduce the number of inconsistencies in the model. Finally, these combinations are presented to the user so that he can choose which one he wants to apply to the model.

d) Key concepts

If one wants to apply the current approach one needs to define the set of cause detection rules and the generator functions for a given meta-model. Notice that both kinds of rules are independent. Whereas the cause detection rules define new kinds of inconsistencies to be detected, the generator functions define new ways to change the model in order to correct them.

e) Analysis of the approach

The following lines analyze the present approach according to our previously defined criteria.

Flexibility features:

- The approach has *multiple coverage*. The approach tries to find a resolution to fix all inconsistencies in the model but it only guarantees that the repaired model has less inconsistencies than the initial one.
- The approach is *non monotonic* because new inconsistencies might be introduced by resolutions. The approach only makes sure that the repaired model has less inconsistencies than the initial one, but no guarantees can be made on which inconsistencies are present in the final model.
- The resolution space is *multiple* because it presents all found solutions to the user.
- The approach is *language independent* because it is based on a meta-model neutral approach.

Usability features:

- The approach does not have *any* selection capability.
- The approach *has tool integration* as the approach has been integrated with two modelling tools, namely Papyrus and the UML2 Tools.

- *No intervention* of the user is allowed during the computation of resolutions.

Extensibility features:

- The approach supports *no* solver extensibility as the Prolog engine that it uses is fixed.
- The approach has *resolution* extensibility since new resolutions can be added by the means of the generator functions.
- The approach has *semi-automatic* inconsistency rule extensibility because generator functions need to be manually updated when new inconsistency rules dealing with elements that are not covered by the existing functions are added.

2.3.2 Egyed's Approach

a) Description of the approach

Egyed [35, 128, 161] proposes an approach for assisting designers in fixing inconsistencies in UML models. In this approach, inconsistency rules are black boxes (Java programs) whose behaviour is observed at runtime.

When an inconsistency rule is executed, it returns one *rule instance* for each identified inconsistency. A *rule instance* references the model elements that have been visited during the execution of a rule.

The main hypothesis of this approach is that any inconsistency can be fixed by changing only one of the references of the corresponding *rule instance*. The principle of the approach is then to identify all possible changes and to provide to the user the ones that solve the inconsistency.

b) Overall process

The overall process is illustrated by the following pseudo-code:

Input : A model m, a set of inconsistencies I and the last change c
Steps :

1. User selects an inconsistency $i \in I$
2. Approach filters out the rule instances that do not refer to elements that have been changed by c
3. Approach uses the generator functions to list all possible changes that can be performed on elements that have been changed by c
4. Approach filters out the changes that add new inconsistencies as well as the ones that do not fix the selected inconsistency
5. User selects one of the remaining changes

Output : A model m' with a set of inconsistencies $I' = I \setminus \{i\}$

This approach takes as input a model, a change and a set of inconsistency rule instances. Since only one inconsistency may be resolved at a time, the approach asks the user to select one. The objective of the approach is, at this point, to propose changes to the model that do not generate new inconsistencies and that fix the selected one.

Changes to any part of the model included in a rule instance can have an impact on the consistency of the model. The approach then reduces the scope of this search by filtering out the model elements that were not part of the change that caused the inconsistency. The underlying assumption is that the last changed elements (just before the introduction of the inconsistency) are the cause of the inconsistency. Then, only these model elements have to be changed to solve the inconsistency.

As there may be an infinite set of changes that can be applied to a model element, the approach considers only a finite subset of them. To face this issue, the so-called *generator functions* are introduced. These functions generate a finite set of possible changes for any given model element.

The last step identifies the impact of each candidate change to the consistency of the system. The approach then performs the candidate changes and checks all the inconsistency rules in order to keep only changes that do not introduce new inconsistencies. The final candidate changes are then presented to the user who takes the final decision on which of them should be executed.

c) Example

Regarding our example, three black boxes correspond to the three inconsistency rules R_1, R_2 and R_3. When they are executed, three *rule instances* are returned, which correspond to I_1, I_2 and I_3.

Let us consider that the *rule instance* corresponding to I_1 at least references six parts of the model: the message name `play`, the lifeline pointed by the message, the class of the lifeline, and the 3 operations owned by this class. We also consider that the last change, which introduced I_1, targets the message name `play`.

In our example, the generator function corresponding to the name attribute of messages proposes only names that match the names of the operations owned by the message's target class. The generator function then proposes three changes, one for each of the three operations of the `Streamer` class. The one that changes the name to `stream` does not introduce new inconsistencies. It is then proposed to the user who can decide to apply it.

d) Key concepts

To apply Egyed's approach one has to define inconsistency rules as black boxes implemented in a programming language (only Java is currently supported). Moreover, generator functions have to be defined for all the model element kinds. It should be noted that these two key concepts (black boxes and generator functions) are independent. Moreover new inconsistency rules and new generator functions can be defined once the approach has been deployed.

e) Analysis of the approach

Egyed's approach has the following features for our three comparison criteria:

Flexibility features:

- The coverage is *single* because the approach only fixes one inconsistency at a time.
- The approach is *monotonic* because all changes that introduce new inconsistencies are excluded from the proposed solutions.
- The solution space is *multiple*, because many solutions for one given inconsistency are presented to the user at the same time.
- There is *no language independence*, because the approach is based on UML.

Usability features:

- The approach allows the user to select the *inconsistency* he wants to fix. Selection of a subpart of the model is not supported.
- The approach has *tool integration* because it has been integrated into the IBM Rational Rose and Rational Software Modeler tools.
- The approach has *no user intervention* since the user cannot guide the computation of the proposed resolutions.

Extensibility features:

- The approach has no *solver extensibility* because the solving algorithm is provided by the approach and cannot be changed.
- The approach has *resolution* extensibility since new resolutions can be added by means of the generator functions.
- The approach has *semi-automatic* inconsistency rule extensibility because resolutions for new inconsistencies are computed automatically by means of the generator functions, but if new inconsistency rules deal with elements that are not covered by the existing generator funcions they need to be adapted manually.

2.3.3 Kleiner's Approach

a) Description of the approach

Kleiner [70] proposes an approach that fixes inconsistencies using constraint solving. In this approach, the user defines inconsistency rules using the OCL+ language, inspired by OCL.

The approach then transforms a model, its meta-model and the OCL+ rules to an instance of a constraint satisfaction problem (CSP) or a boolean satisfiability problem (SAT). The approach uses an external solver to solve the CSP or SAT problem. The output of the solver is translated back to an updated model, that is entirely inconsistency-free. It should be noted that only changes that add elements to the model can be considered by the solver. Existing model elements cannot be removed and properties cannot be modified.

b) Overall process

The process is illustrated by the following pseudo-code:

Input: A model m with a set of inconsistencies I.
Steps:

1. Approach translates the model and the inconsistency rules to the solver input format (CSP or SAT)
2. Approach asks the solver to find a solution
3. Approach converts the solution to an updated model
4. User accepts the proposed model

Output: A model m' with no inconsistencies.

c) Example

In our example, the three inconsistency rules have to be expressed in OCL+. SAT is then chosen by the user. The model, its meta-model and the OCL+ rules are translated into a SAT instance. Finally, the SAT solver is called in order to solve this SAT instance. The solution is then converted to an updated model.

In our example, the updated model has a new operation, named `play`, in the class `Streamer` to solve I_1. To solve I_2 it also has a new reflexive transition, named `play`, on the state `waiting`. Finally, it has a new association from `Streamer` to `Display` to solve I_3. Those changes do not introduce any new inconsistency.

d) Key concepts

In order to apply Kleiner's approach, the inconsistency rules need to be expressed in OCL+. Moreover, in order to use the approach for a new kind of model, a new translator needs to be provided that inputs models of this kind and outputs corresponding solver constraints.

e) Analysis of the approach

Kleiner's approach has the following features according to the three comparison criteria:

Flexibility features:

- The approach's coverage is *full* as all inconsistencies are fixed in a step.
- The approach is *monotonic* as it does not introduce new inconsistencies.
- The approach returns *one* single solution.
- The approach is *language independence*, because it is based on the meta-metamodel Eclipse Modeling Framework metamodel (Ecore)[2]. The approach can use all modeling languages that have Ecore as meta-metamodel.

[2] `http://wiki.eclipse.org/Ecore`

Usability features:

- The approach does not have *any* selection capability.
- There is no *tool integration* as the approach is not integrated in any CASE tool.
- There is no *user intervention* possible during the resolution process.

Extensibility features:

- The approach has *solver extensibility* because it is solver-independent and can use any external solver to solve the CSP or SAT problem.
- The approach has *no resolution extensibility* because new resolutions cannot be added.
- The approach has *automatic inconsistency rule extensibility* because new inconsistencies are automatically handled by their fixing algorithm.

2.3.4 Mens's Approach

a) Description of the approach

Mens [97, 98] proposes an incremental approach to model inconsistency resolution based on graph transformation. On top of AGG, a general-purpose graph transformation tool, an interactive prototype tool was developed to select and apply conflict resolution rules. In this approach, models are represented as typed, attributed graphs, and the metamodel is represented as a typed graph to which the model graphs need to conform.

Model inconsistencies are detected through simple graph transformation rules, consisting of a left-hand side that is used to match the presence and/or absence of graph patterns that correspond to the inconsistency, and a right-hand side that simply adds to the graph a new *Conflict* node that is connected to the nodes of the graph that are the source of the detected inconsistency.

Inconsistency resolution rules are expressed as graph transformation rules that modify the structural patterns that are the source of the inconsistency. For the same type of inconsistency, multiple alternative resolution rules may be specified. The left-hand side of these rules contain the *Conflict* nodes added during the inconsistency detection phase. The right-hand side removes these *Conflict* nodes and modifies the graph in such a way that the inconsistency is no longer present.

b) Overall process

The approach follows an interactive and iterative inconsistency resolution process, as illustrated below. The iterative process ends when the user is satisfied or when all inconsistencies have been resolved.

Input: A model m with a set of inconsistencies I.
Steps:

1. Approach applies all inconsistency detection rules to m to add *Conflict* nodes (each corresponding to an inconsistency $i \in I$).

2. User selects one of the *Conflict* nodes to fix i.
3. Approach proposes set of resolution rules to choose from to fix i.
4. User selects one of the resolution rules to apply, and provides additional input if required by the resolution rule.
5. Approach applies the resolution rule that fixes i.
6. Go to step 1.

Output: A model m' with a set of inconsistencies I' where $I \neq I'$

c) Example

For the running example, three graph transformation rules would need to be written to detect the inconsistencies specified by the rules R_1, R_2 and R_3. Applying these rules on the model will create 3 *Conflict* nodes that represent the inconsistencies I_1, I_2 and I_3. It is up to the user to select which one is to be resolved first. Let us imagine that the user selects I_1. Let us also imagine that two alternative resolutions are specified as graph transformation rules: one that removes a message from the sequence diagram if there is no corresponding operation in the class diagram, and one that adds a new operation (with the same name as the message) in the class diagram. These two resolutions are presented to the user, who decides to select one of the two. This resolution is then applied, removing the inconsistency I_1, and the updated model together with the remaining inconsistencies I_2 and I_3 is presented again to the user.

d) Key concepts

To use Mens's approach, one needs to define graph transformation rules for detecting the inconsistencies, and several graph transformation rules to specify how each inconsistency should be resolved. Multiple alternative resolutions rules can be defined for the same inconsistency. In this case, during the inconsistency resolution process, the user might be asked to select which inconsistency resolution to apply.

The novelty of the approach, compared to other approaches is the use of critical pair analysis to formally analyze whether and how a resolution rule may potentially introduce new inconsistencies. This is needed, since the approach does not guarantee that the resolution of an inconsistency does not introduce new inconsistencies.

e) Analysis of the approach

Mens's approach has the following features according to the three comparison criteria:

Flexibility features:

- The tool has either *single coverage* or *full coverage*, depending on how it is used. In the first scenario of use, the user detects a particular type of model inconsistency, and selects and applies one of the inconsistency resolution rules to resolve this inconsistency. In the second scenario, the tool can non-deterministically apply all detection rules and resolution rules. (In case of multiple matches of a given rule, or multiple rules to choose from the tool will randomly pick one of them.)

- The approach is *non-monotonic* since the resolution of an inconsistency may introduce new inconsistencies.

- Every execution of the resolution approach will give rise to a *one solution*, but multiple runs may provide multiple solutions because of the non-deterministic resolution process.

- The approach is *language independent* since it is based on graphs and type graphs. If a particular language needs to be supported, its metamodel needs to be expressed as a type graph, and the models need to be translated into graphs.

Usability features:

- The approach allows the user to select only a subset of *rules* by ignoring or disabling certain detection and/or resolution rules. Since the approach is interactive, each *inconsistency* needs to be selected by the user in order to resolve it. The approach does not allow the user to restrict to a subpart of the model.

- The approach is *not integrated* into any CASE tool. Since it is a research prototype, the AGG graph transformation tool has been directly extended with a dedicated user interface for selecting and applying resolution rules.

- *User intervention* is supported if the user follows the manual interactive scenario: he interacts with the tool to select between different alternative resolution rules, and to provide input parameters to particular resolution rules.

Extensibility features:

- The approach has *no solver extensibility* because the graph transformation engine provided by AGG can not be changed.

- The approach has *resolution extensibility*, the existing detection and resolution rules can be modified and new resolution rules can be added manually, by adding new graph transformation rules to AGG's graph grammar. This requires the user to know the syntax and semantics of graph transformation and the metamodel.

- To generate resolutions to new inconsistency rules, new resolution rules need to be added *manually* and explicitly.

2.3.5 Nentwich's Approach

a) Description of the Approach

Nentwich [110] proposes a framework that detects and fixes inconsistencies within distributed XML documents. Inconsistency rules are specified thanks to a proprietary language that is based on XPath.

The main component of the framework is the *repair administrator*. This component is responsible for both detection and resolution. It downloads a set of XML documents and checks them against a set of inconsistency rules. When it detects an inconsistency it generates the set of actions that repairs it thanks to pre-defined logical mappings.

b) Overall process

The overall process is illustrated below:

Input: A model m with a set of inconsistencies I.
Steps:

1. User selects an inconsistency $i \in I$
2. Approach identifies the logical rule r that detects i
3. Approach uses the logical mapping to map r and m in order to generate a set of changes C' that fix i
4. User selects one of the proposed changes in C'

Output: A model m' with a set of inconsistencies I' where $i \notin I'$

This approach takes as input a model with inconsistencies. It computes the resolutions thanks to pre-defined mappings that map inconsistency rules to repair actions. In order to use these mappings, the approach has to identify which of the inconsistency rules detect the inconsistencies. The repair actions are provided to the user, who takes the final decision on which of them is going to be applied.

c) Example

In our example, the three inconsistency rules R_1, R_2 and R_3 have been expressed with the proprietary language. The *repair administrator* has downloaded the model (presented in XMI) and has detected the three inconsistencies I_1, I_2 and I_3.

Suppose that the user selects I_1. The approach identifies that I_1 has been detected thanks to R_1. It then uses the pre-defined mappings to generate the repair actions. Suppose that the three following examples are proposed: (i) deleting the `play` message, (ii) changing its name to match one of the operations in the class `Streamer` and (ii) renaming some operation in the class `Display` to `play`. The user then has to select one of them.

d) Key concepts

In order to apply Nentwich's approach, one needs to define its inconsistency rules as logical formulas following their proprietary XML-based representation. Repair actions are computed automatically by a repair administrator tool.

e) Analysis of the approach

The following lines analyze the present approach according to our previously defined criteria.

Flexibility features

- The coverage is *single* because the repair administrator only allows fixing one inconsistency at a time.

- The approach is *non monotonic* because every inconsistency is guaranteed to be fixed by every fix proposed by the repair administrator. However, there is no guarantee that the fix for one inconsistency is not going to add other inconsistencies.

- The resolution space is *multiple*, because the multiple fixes are computed and presented to the user who then chooses the one that is going to be applied to the model.

- The approach *has language independence* because it works with any XML based data format.

Usability features

- The approach allows the user to select the *inconsistency* he wants to fix and the set of *rules* that are applicable. The approach cannot select a subpart of the model.

- The approach has *tool integration* because it has been integrated in one prototype modeling tool.

- The approach has *no user intervention* because the user cannot guide the process of finding new resolutions.

Extensibility features

- The approach has *no solver extensibility* because the solving algorithm can not be changed

- The approach has *no resolution extensibility* because new resolutions cannot be added.

- The approach has *automatic inconsistency rule extensibility* because new inconsistencies are automatically handled by their fixing algorithm.

2.3.6 Van Der Straeten's Approach (I)

a) Description of the approach

Van Der Straeten [152, 154] proposes an approach to model inconsistency resolution rules thanks to a dedicated query and rule language, named *nRQL*. This language is designed to query knowledge bases expressed using Description Logics (DLs). DL is a family of logic languages used in ontology engineering. Models are represented as sets of logic facts that define model elements.

Model inconsistencies are detected by nRQL queries. Variables in these queries are bound to model elements that satisfy the query expression. Model resolutions are expressed by nRQL rules. Their premise contains the query expressing a particular inconsistency rule. Their conclusion states how to resolve the corresponding inconsistencies. A resolution consists of a sequence of statements, where each statement is responsible for either adding or removing data to/from the model.

b) Overall process

The overall process is illustrated below:

Input: A model m with a set of inconsistencies I.
Steps:

1. User selects an inconsistency $i \in I$ to fix
2. Approach proposes a set of nRQL rules in order to choose from to fix i
3. User selects one of the rules and provides additional input required by the rule
4. Approach applies the rule that fixes i
5. User selects another inconsistency (step 1)

Output: A model m' with a set of inconsitencies I' where $I \neq I'$

This approach follows an interactive and iterative process. First, all model inconsistencies are computed for the input model by applying the nRQL logic queries. Next, the user can select any of the detected inconsistencies, and choose to apply one of the nRQL rules dedicated to this inconsistency. It produces an updated model that becomes the new input model for the resolution. The iterative process continues until the user is satisfied or until all inconsistencies have been resolved.

c) Example

With our example, three logic queries are written to detect the inconsistencies specified by the rules R_1, R_2 and R_3. Using these queries on the model will find the inconsistencies I_1, I_2 and I_3. It is up to the user to select which one is to be resolved. Let us imagine that the user selects I_1. Let us imagine that two different logic resolution rules have a premise that detect I_1. One rule has a conclusion that removes the `play` message from the sequence diagram. The other one adds a `play` operation to the class `Streamer`. These two resolutions are presented to the user, who needs to select one of them. The chosen resolution is then applied. The updated model together with the remaining inconsistencies are presented again to the user.

d) Key concepts

To apply Van Der Straeten's approach, inconsistency rules need to be expressed using the nRQL language. Furthermore, one needs to implement inconsistency resolution rules. These rules reference the inconsistencies that they resolve and provide statements that change the model in order to resolve the inconsistency. Multiple rules can be defined for the same inconsistency. In that case it is up to the user to select the rule to apply.

e) Analysis of the approach

The following lines analyze the present approach according to our previously defined criteria.

Flexibility features:

- The approach has *full coverage*, since resolutions are proposed to the user for all the detected inconsistencies at a time.
- The approach is *non monotonic* because it is possible to introduce new inconsistencies during resolution.
- The approach can return *multiple solutions* which are executed one at a time. Rules can require user input and when multiple rules are applicable, the user must choose the rule he wishes to apply.
- The approach is *language independent* as long as the models can be expressed in the underlying logic.

Usability features:

- The user can select one or a set of detection and resolution *rules* and the user can select a specific resolution to execute. Because the approach has full coverage, the user cannot select one or several *inconsistencies* he wants to resolve. The approach does not allow to select a part of the model under consideration.
- The approach has been *integrated* as a plugin in one modeling tool (Poseidon).
- *User intervention* is supported by the approach because the user needs to select a particular resolution and it could be that the user needs to give input if necessary to the execution of the corresponding resolution rule.

Extensibility features:

- The approach has *no solver extensibility* because the rule engine provided by RACER can not be changed.
- The approach has *resolution extensibility*, the existing rules cannot be changed but new ones can be added manually. However, in order to do that, the user needs to know the rule language nRQL, the detection rules and the translated metamodel.
- To generate resolutions to new inconsistency rules, new resolution rules need to be added *manually* and explicitly.

2.3.7 Van Der Straeten's Approach (II)

a) Description of the approach

In Van Der Straeten [156] the usage of the model finder *Kodkod* is explored to resolve inconsistencies. The inconsistency resolution process can be automated using *Kodkod* without the need of manually writing any inconsistency resolution rule or resolution generation functions. *Kodkod* is a constraint solver. The logic accepted by *Kodkod* is a core subset of the Alloy modelling language supporting first order quantifiers, connectives, arbitrary-arity relations and transitive closure.

Models are expressed as *Kodkod* problems. A *Kodkod* problem consists of a universe declaration, i.e., a set of atoms, a set of relation declarations and a formula. The universe of a *Kodkod* problem representing a UML model contains an atom for each model element. A relation in a *Kodkod* problem is declared through a relational variable name, its arity

and bounds on its value. *Kodkod* requires the relational variables to be bound prior to analysis. Every relational variable must be bound from above by a relational constant, a fixed set of tuples drawn from the universe of atoms. Each relation must also be bound from below by a relational constant, i.e., a lower bound containing the tuples that the variables value must include in an instance of the formula. The union of all relations lower bounds forms a problems partial instance. Each UML metaclass is expressed as a unary relation. Its value is the set of model elements that represent its instances. The UML meta-association ends and attributes are translated into the corresponding k-arity relational variables. The values of these relational variables are tuples containing the UML model elements involved in the corresponding UML meta-associations or meta-attributes. In order to generate consistent models w.r.t. a consistency rule, the consistency rule is specified as part of the *Kodkod* problems formula. *Kodkod*s analysis will search for an instance of the consistency within the provided model.

b) Overall process

The overall process is illustrated below:

Input: A model m with a set of inconsistencies I and the corresponding consistency rules.
Steps:

1. User selects an inconsistency $i \in I$ to fix and a place where to fix it,
2. Approach proposes a set of models in which i is resolved,
3. User selects one model
4. User selects another inconsistency (step 1)

Output: A model m' with a set of inconsistencies I' where $I \neq I'$

This approach can be used as an interactive and iterative process. The approach assumes that model inconsistencies can be detected and that the model elements involved in the inconsistency are known. The user can select an inconsistency and a model element involved in the inconsistency. Based on this information, a *Kodkod* problem is generated and analysis of this problem is done. As a result, a set of models in which the selected inconsistency is resolved are generated and presented to the user. The user can select a model and this model becomes the new input model for resolution. The iterative process continues until the user is satisfied or until all inconsistencies have been resolved.

c) Example

The approach assumes that an inconsistency detection approach has detected the inconsistencies I_1, I_2 and I_3. It is up to the user to select which one needs to be resolved. Let us imagine that the user selects I_1 and as a possible location for resolution the operation `play`. After translation into *Kodkod* and the analysis, three different solutions will be presented to the user, one model where the operation is `stream`, another where the operation is `wait` and one where the operation is `connect`. The user can select one of these models. The updated model together with the remaining inconsistencies are presented again to the user.

d) Key concepts

To apply this approach, the model and the consistency rule need to be expressed as a *Kodkod* problem. Furthermore, the user can also select the location, i.e., a model element where the inconsistency can get resolved. No inconsistency resolution rules or generator functions need to be implemented.

e) Analysis of the approach

The following lines analyze the present approach according to our previously defined criteria.
Flexibility features:

- The approach has *single coverage*, since resolutions are proposed to the user only for one inconsistency at a time. However, as specified in [156], the approach can be extended easily to considering multiple inconsistencies.
- The approach is *non monotonic* because it is possible to introduce new inconsistencies during resolution.
- The approach returns *multiple solutions*, i.e., multiple models are generated by *Kodkod*.
- The approach is *language independent* as long as the models can be expressed as *Kodkod* problems.

Usability features:

- The user needs to select *one inconsistency* he wants to resolve.
- The translation of the models is done automatically from within an Eclipse plugin. However the selection of the models is *not integrated* in a modeling tool.
- During resolution, user intervention is *not* possible.

Extensibility features:

- The approach has *solver extensibility*, *Kodkod* can use different SAT solvers.
- The approach has *no resolution extensibility* because new resolutions cannot be added.
- The approach has *automatic inconsistency rules extensibility* because new inconsistencies are automatically handled by their fixing algorithm.

2.3.8 Xiong's Approach

a) Description of the approach

Xiong [164] proposes an approach to automatically fix inconsistencies within models. In this approach, inconsistency rules are written in a proprietary language, named *Beanbag*.

Beanbag rules have two semantics. One is dedicated to inconsistency detection whereas the other one is dedicated to inconsistency resolution.

Both semantics take as input a model and a change performed on it. Regarding inconsistency detection, the rule detects if the change introduces new inconsistency. Regarding inconsistency resolution, the rule modifies the model with the objective to compensate for the inconsistency, solving it.

Once an inconsistency is detected by a rule, the approach uses the resolution semantics of the same rule in order to produce a resolution for the inconsistency.

b) Overall process

The overall process is illustrated below:

Input: A model m with a set of inconsistencies I and last changes C
Steps:

1. User selects an inconsistency $i \in I$
2. Approach identifies one Beanbag rule r whose detection semantics detects i
3. Approach uses the resolution semantics of r to generate changes C' that fix i from m and C
4. User applies the proposed changes in C'

Output: A model m' with a set of inconsistencies I' where $i \notin I'$.

The approach does not make a clear separation between inconsistency detection and inconsistency resolution. Each time the user wants to fix an inconsistency, the approach identifies which of the Beanbag rules detects the inconsistency. Once the rule has been identified, its resolution semantics is used in order to generate the changes. The user has then to accept the proposed changes.

The approach does not guarantee that the resolution generated by one inconsistency rule does not introduce new inconsistencies, detected by other rules. This is due to the fact that each Beanbag rule, including its detection and resolution semantics, considers only one inconsistency.

c) Example

Regarding our example, three Beanbag rules correspond to the inconsistency rules R_1, R_2 and R_3.

Consider the fixing of inconsistency I_1. Suppose that this inconsistency was introduced by the user renaming a `stream` message in the sequence diagram into `play`. Once I_1 has been selected by the user, the approach identifies that it has been detected by R_1. As a consequence, the resolution semantics of R_1 is used in order to fix I_1. A possible resolution semantics would be to replace the message's name by the old name `stream`.

d) Key concepts

If one wants to apply this approach, one needs to (re)write the inconsistency rules as Beanbag rules. Extra care needs to be taken in order to make sure that the resolution semantics of these rules will produce the desired corrections. As the authors state in [164], this is a trial-and-error process that should be executed by the metamodeler.

e) Analysis of the approach

Xiong's approach has the following features according to our comparison criteria:

Flexibility features:

- The coverage is *single* because the approach only considers a single inconsistency at each step.
- The approach is *non monotonic*, because resolutions to one rule may cause inconsistencies to others.
- The solution space explored by this approach produces only *one solution* for each inconsistency.
- As far as we can derived from the information from the article, the approach has *no language independence* because it is based on models described in a proprietary language based on dictionary data structures.

Usability features:

- There is *no selection* support.
- There is no *tool integration* for this approach.
- There is no *user intervention* for this approach.

Extensibility features:

- The approach has *no solver flexibility* because the solver is provided by the language runtime and cannot be changed.
- This approach allows the definition of new resolution rules only by *manually* changing its Beanbag code.
- The approach has *semi-automatic* inconsistency rules flexibility because inconsistency rules are obligatorily defined when new inconsistency rules are defined, but they user needs to manually make sure that its resolution semantics works as expected.

2.4 Summary of the Study

In this chapter we have carried out a feature-based analysis of design model inconsistency resolution approaches. We used this feature-based approach to study eight recent and representative resolution approaches. We focused on three criteria of interest: flexibility, usability and extensibility. Tables 2.1, 2.2 and 2.3 summarise the study of all considered approaches against the three main study criteria.

The features presented in this chapter were used to identify the strengths and weaknesses of each approach, but these features cannot be used to compare and classify the approaches between them. Because each user can have particular needs, preferences, desires and interests, an interesting feature for a user is not necessarily of interest for another

user. For example, a user that only wants to resolve inconsistencies in UML models, may not be interested in the language independence of certain approaches because he only wants to resolve inconsistencies in one language (UML). Controlled user studies would be needed to assess what approach is the most suitable in practice, for each particular need. This study is out of the scope of this dissertation.

Even if a comparison between the approaches is not possible, the features can be used to reveal common features between the approaches.

It can be seen in Table 2.1 that 5 out of 8 approaches have single coverage; that most of the approaches (6 of 8) are non monotonic; that the approaches return multiple solutions for 5 of 8 approaches and return one solution for 3 of 8 approaches; and that 6 out of 8 approaches are language independent.

It can be noticed in Table 2.2 that there is a lot of variation in the approaches w.r.t. the selection feature. 3 approaches do not propose any selection to the user, 4 approaches propose to the user the selection of the inconsistency, only 2 approaches propose the selection of rules and none of the approaches proposes to the user the selection of a part of the model. Notice also that only half of the analysed approaches are integrated into a CASE tool and only two approaches allow user intervention.

It can be seen in Table 2.3 that only two approaches support multiple solvers; 5 out of 8 approaches are resolution extensible; and that there is no common pattern in the approaches w.r.t. the support of automatic inconsistency rules. 3 approaches have automatic inconsistency rule extensibility, 3 approaches have semi-automatic extensibility and 2 approaches have manual extensibility.

	Considered features			
Approach	Coverage	Monotonicity	Resolution space	Language Independence
Almeida da Silva	multiple	no	multiple	yes
Egyed	single	yes	multiple	no
Kleiner	full	yes	one	yes
Mens	single or full	no	one	yes
Nentwich	single	no	multiple	yes
Van Der Straeten I	full	no	multiple	yes
Van Der Straeten II	single	no	multiple	yes
Xiong	single	no	one	no

Table 2.1 – Summary of model inconsistency resolution approaches for the *Flexibility* criterion

	Considered features		
Approach	Selection	Tool integration	User intervention
Almeida da Silva	no	yes	no
Egyed	inconsistency	yes	no
Kleiner	no	no	no
Mens	inconsistency, rule	no	yes
Nentwich	inconsistency, rule	yes	no
Van Der Straeten I	rule	yes	yes
Van Der Straeten II	inconsistency	no	no
Xiong	no	no	no

Table 2.2 – Summary of model inconsistency resolution approaches for the *Usability* criterion

2.5 Discussion

Several approaches have been proposed to resolve model inconsistencies. We will review the different approaches, highlighting their weaknesses. We will propose a new approach avoiding these weaknesses.

In Mens [97,98] and in Van Der Straeten I [99,135,154] the authors specify resolution rules *manually*, which is an error-prone process. *Automatic* generation of inconsistency resolution actions aims to resolve this problem. Nentwich [109,110] achieve this by generating resolution actions *automatically* from the inconsistency rules. The execution of these actions, however, only resolves one inconsistency at a time. As recognised by the authors, this causes problems when inconsistencies and their resolutions are interdependent. Mens *et al.* [97] propose a formal approach based on graph transformation to analyse these interdependencies.

Kleiner [70] proposes an approach that fixes inconsistencies using constraint solving. They propose a new language, named OCL+, to express inconsistency rules. Their approach can use any solver (such as SAT or CSP) in order to identify how to change the inconsistent model in order to make it compliant with all the constraints. The approach proposes only one resolution to the inconsistent model. The resolution proposed by this approach cannot remove existing model elements and properties cannot be modified.

Xiong [164] define a language to specify inconsistency rules and the possibilities to resolve the inconsistencies. This requires inconsistency rules to be annotated with resolution information. Almeida da Silva [2] propose an approach to generate resolution plans for inconsistent models, by extending inconsistency detection rules with information about the causes of the inconsistency, and by using manually written functions that generate resolution actions. In both approaches inconsistency detection rules are polluted with resolution information.

Instead of explicitly defining or generating resolution rules, a set of models satisfying a set of consistency rules can be generated and presented to the user. Egyed [33,35,128,161] define such an approach for resolving inconsistencies in UML models. Given an inconsistency and using choice generation functions, their approach generates possible resolution choices, *i.e.*, possible consistent models. The choice generation functions depend on the modeling language, *i.e.*, they take into account the syntax of the modeling language, but they only consider the impact of one consistency rule at a time. Furthermore these choice

	Considered features		
Approach	Solver	Resolution	Automation
Almeida da Silva	no	yes	semi-automatic
Egyed	no	yes	semi-automatic
Kleiner	yes	no	automatic
Mens	no	yes	manual
Nentwich	no	no	automatic
Van Der Straeten I	no	yes	manual
Van Der Straeten II	yes	no	automatic
Xiong	no	yes	semi-automatic

Table 2.3 – Summary of model inconsistency resolution approaches for the *Extensibility* criterion

generation functions need to be implemented manually.

In Van Der Straeten II [156] we use *Kodkod*, a SAT-based constraint solver using relational logic, for generating consistent models *automatically*. While the approach guarantees correctness and completeness (within the considered lower and upper bounds of the relations defined in the problem), a major limitation is its poor performance and lack of scalability.

The approach that we will propose in Chapter 4 intends to overcome the aforementioned shortcomings by *automatically* generating *multiple* inconsistency resolutions for resolving *multiple* inconsistencies at the same time in a *scalable* way.

3

Introduction to Automated Planning

As aforementioned, in Chapter 1, automated techniques to resolve model-driven inconsistencies are essential. We propose to use the artificial intelligence technique of automated planning for this purpose. In this chapter we introduce automated planning, and present and formally define classical planning. A classical planning is composed of a representation language and an algorithm. We describe them and introduce the most common representation languages and algorithms. We conclude this chapter by presenting some classical planning implementations.

3.1 Introduction

Planning is the human capability of reasoning before acting. It is an explicit and deliberate process that chooses and organizes actions by anticipating their effects. This is a complex intellectual capability because the world changes continuously and we have a limited vision of it [51, 65].

Automated planning is an *Artificial Intelligence* branch that studies and puts into practice this explicit and deliberate process in a computational way. The principal motivation of automated planning is the study and design of autonomous intelligent machines. Automated Planning emerged in the late fifties (1950s) from converging investigations into state-space search, theorem proving, and control theory to answer the needs of domains like robotics and scheduling. The STRIPS[1] [43] planner was developed in 1971 to control the behavior of the autonomous robot SHAKEY, the first general purpose robot developed by the Artificial Intelligence laboratory at the Stanford Research Institute (Figure 3.1). STRIPS is considered as the first major planning system. Since the introduction of STRIPS, an automated planning approach is defined as a program that generates a *plan*, *i.e.*, a sequence of actions that lead from an initial state to a state meeting a specific predefined goal [51, 65, 129].

Figure 3.1 – SHAKEY: The first general purpose robot.
Source and Copyright – Artificial Intelligence laboratory at the Stanford Research Institute.

There is a wide range of domains in which planning can be applied. For example: path and motion planning, perception planning and information gathering, navigation planning, manipulation planning, communication planning, social and economic planning, urban planning, family planning, financial planning [51].

To deal with these diverse forms of planning, we can use either *domain-specific* or *domain-independent* planners. Domain-specific planners use a specific representation and techniques adapted to the problem. In contrast, domain-independent planners are generic planners that take as input the problem specifications and the knowledge about their domain [51]. For many applications, domain-specific planners are crucial because they

[1]STRIPS stands for STanford Research Institute Problem Solver.

exploit the problem and domain specifications for efficiency reasons. Domain-independent planners on the other hand, can be used in many domains and for many problems.

Automated planning has been used with success in different and demanding application domains, such as: game AI, robotics, the Hubble space telescope, The Deep Space 1 [8, 23, 66, 69, 107]. The research in automated planning has been active and maturing in artificial intelligence research since the late fifties, and many papers on automated planning are published in artificial intelligence journals and dedicated conferences (*e.g.* International Conference on Artificial Intelligence Planning and Scheduling (ICAPS)[2]).

3.2 Classical Planning

Classical Planning is an automated planning variant that aims to find a sequence of actions that reaches a desired state in a *Finite*, *Static*, *Deterministic*, *Implicit time* and *Fully observable* world. This means that the world has a finite set of states, the world stays in the same state until a new action is executed and the execution of an action is instantaneous and brings the world into a single other state. There is also complete knowledge about the current state of the world.

Each classical planning approach consists of: a *representation language* used to describe the *problem domain* and the *specific problem*; an *algorithm* describing the mechanism to solve the problem; and a sequence of *generated plans* produced as output.

Other variants of automated planning relax the world's assumptions in order to act in real world problems. For example: *Temporal Planning* [42, 88] aims to address problems where the *Implicit time* assumption is not true anymore, that means when the actions' effects may not be instantaneous; *Planning Under Uncertainty* [28, 160] aims to resolve problems in a *nondeterministic* and *partially observable* world, that means that the planner has an incomplete knowledge of the current state and that the actions' effects may be stochastic.

In this dissertation we are only interested in problem of classical planning, because the inconsistency resolution in design models fulfills the classical planning assumptions. Also, most of the state-of-the-art in automated planning is focused on classical planning. This interest sparked off well formalized and defined problems with algorithms and techniques that scale-up reasonably well [65].

3.3 Formal Definition of Classical Planning

Let Π be a finite set of all possible logic predicates, and S be a finite set of states such that $\forall s \in S : s \subseteq \Pi$.

The problem of classical planning can be formally defined [65] as a tuple $CP = (PD, SP, ALG)$. The problem domain PD is expressed as a set of possible actions A and the specific problem is defined as a tuple $SP = (s_0, dg)$:

- $s_0 \in S$ is the initial state;
- A is a finite set of actions. Each action $a \in A$ is a function $a : S \rightarrow S : s \rightarrow s'$ with $s' \neq s$;

[2]Since 2003, the International Conference on Automated Planning and Scheduling (ICAPS) is the merger of International Conference on Artificial Intelligence Planning Systems (AIPS) and European Conference on Planning (ECP).

- $ALG : S \rightarrow A : s_i \rightarrow a_i$ is the algorithm that chooses the action to be applied in the state s_i;
- $dg \subseteq \Pi$ is a partially specified state that describes the desired goal.

Using this definition, finding a solution for the classical planning problem CP consists in generating a sequence of actions $Plan = (a_1, a_2, ..., a_n)$ corresponding to a sequence of states $(s_1, ..., s_n)$ obtained from the initial state s_0 by iteratively applying each action a_i to state s_{i-1} until s_n where $dg \subseteq s_n$, meaning that the desired goal dg is satisfied.

The optimal *plan* is the one that minimizes the expression $\sum_{i=1}^{n} c(s_{i-1}, a_i)$ [65] where $c : S \times A \rightarrow \mathbb{N}$ is a function representing the cost $c(s, a)$ of applying the actions $a \in A$ in the state $s \in S$.

3.4 The Representation Language

The *representation language* is used to describe the syntax and semantics of classical planning. This language must support the description of the planning domain and the specific problem.

We will use blocks-world in the following part, an example of automated planning borrowed from [129,162]. It consists of a set of blocks on a table and a robot arm used to move the blocks. The robot arm can pick up one block at a time and stack it on top of another block or drop it on the table. The figure 3.2 illustrates a blocks world problem with 3 blocks (A,B,C). The initial state is the blocks A and B on the table and the block C on top of the block A. The desired goal is the block A on top of the other two blocks. The problem is to find a sequence of actions to be performed by the robot arm to go from the initial state to the desired goal.

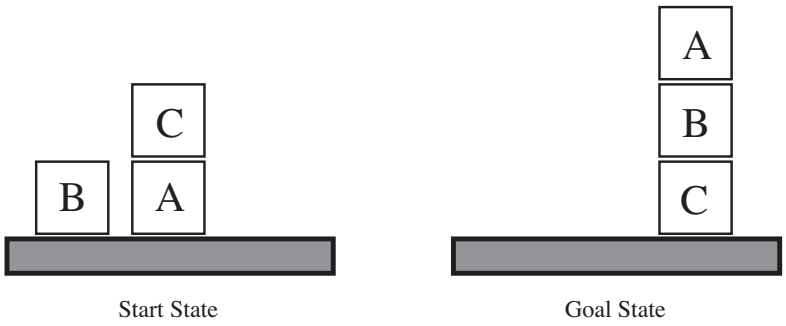


Figure 3.2 – Blocks-world problem.
Source: *Russell and Norvig* [129]

3.4.1 States

Automated planning decomposes the world into logic predicates and represents a *state* as a conjunction of ground[3], functionless[4], non-negated predicates. Classical planning has a closed world assumption meaning that any predicate not mentioned is assumed to be false. In the blocks-world example the logic predicates used are: $On(C, A)$ and $Clear(C)$.

[3] A ground predicate is a predicate without variables.
[4] A functionless predicate is a predicate that cannot be nested.

$On(C, A)$ indicates that the block C is on top of the block A. $Clear(C)$ indicates that there is no block on top of the block C. An example of predicates that cannot be used in a state is: $On(b, x)$ because it is not ground[5]; $On(On(C, A), B)$ because it is not functionless; and $\neg On(B, C)$ because it is a negation.

3.4.2 Problem Domain

The *problem domain* (e.g., blocks-world) is expressed as a *set of possible actions* (e.g., to move a block). A *possible action* specifies a *valid* way to go from one state to another. The action is composed of a *precondition* and an *effect*[6]. The precondition specifies the conditions that must hold in order for the action to be applicable. The effect specifies the changes to be made to the current state. Negated predicates in the effects list represent predicates that are removed from the state. For example, here are the actions to move a block in a blocks-world domain:

$Action: \ Move(b, x, y)$
$Precondition: \ On(b, x) \ \wedge \ Clear(b) \ \wedge \ Clear(y) \ \wedge \ Block(b) \ \wedge \ Block(y)$
$Effect: \ On(b, y) \ \wedge \ Clear(x) \ \wedge \ \neg \ On(b, x) \ \wedge \ \neg \ Clear(y)$

$Action: \ MoveToTable(b, x)$
$Precondition: \ On(b, x) \ \wedge \ Clear(b) \ \wedge \ Block(b)$
$Effect: \ On(b, Table) \ \wedge \ Clear(x) \ \wedge \ \neg \ On(b, x)$

The action $Move(b, x, y)$ moves a block b from the top of x (which can be either a block or a table) to the top of a block y. This action can be executed only if all preconditions are satisfied:

- $Clear(b)$: there is no other block on top of block b;
- $On(b, x)$: block b is on top of x (block or table);
- $Clear(y)$: there is no other block on top of the destination block y;
- $Block(b) \wedge Block(y)$: the object to move b and the destination place y are blocks.

The action's effects are :

- $\neg On(b, x)$: the block b is not anymore on top of x, therefore $On(b, x)$ should be removed from the state representation;
- $On(b, y)$: the block b is now on top of the block y;
- $\neg Clear(y)$: the top of the block y is occupied now by the block b, meaning that it is not clear anymore, therefore $Clear(y)$ should be removed from the state representation;
- $Clear(x)$: the top of x is not occupied anymore by the block b, meaning that it is clear now.

The preconditions and effect are quite similar for the action $MoveToTable(b, x)$.

[5] A literal starting with lowercase is considered as a variable. In this example b and x are variables.
[6] The effect is also sometimes called *postcondition*

3.4.3 Specific Problem

The *specific problem* is expressed by an *initial state* and a *desired goal*. The **initial state** is a special state that represents the current state of the world (*e.g.,* the current configuration of the blocks-world). For example, here are the predicates corresponding to the initial state of the blocks-word problem showed in Figure 3.2:

$On(A, Table) \ \wedge \ On(B, Table) \ \wedge \ On(C, A) \ \wedge$
$Block(A) \ \wedge \ Block(B) \ \wedge \ Block(C) \ \wedge$
$Clear(B) \ \wedge \ Clear(C)$

The **desired goal** is a partially specified state that describes the world that we would like to obtain (*e.g,* the desired configuration in blocks-world). It is a conjunction of predicates; it differs from a normal state in that the predicates could be negative or non-ground. The desired goal is a pattern that can be matched to a state to find a goal. A desired goal for the blocks-world problem can be for example : $On(A, x) \wedge On(x, y) \wedge Block(y)$. This desired goal is to have the block A in top of two other blocks. Two goal states that match the desired goal are :

$On(A, B) \ \wedge \ On(B, C) \ \wedge \ On(C, Table) \ \wedge$
$Block(A) \ \wedge \ Block(B) \ \wedge \ Block(C) \ \wedge$
$Clear(A)$

$On(A, C) \ \wedge \ On(C, B) \ \wedge \ On(B, Table) \ \wedge$
$Block(A) \ \wedge \ Block(B) \ \wedge \ Block(C) \ \wedge$
$Clear(A)$

3.4.4 Languages

Fikes *et al* [43] developed, in 1971, a formal planner approach called *STRIPS*. Its representation language has been more influential than its algorithmic approach [129]. In 1989, Pednault [119] developed a more advanced and expressive language called *Action Description Language*(ADL)[7]. ADL has an improved expressiveness compared to STRIPS. In particular, ADL applies the open-world principle: unspecified predicates are considered as unknown instead of being assumed false. ADL also allows to use negative predicates and disjunctions, whereas STRIPS only allows positive predicates and conjunctions. In recent years, a standard *Planning Domain Definition Language* (PDDL) [89] has been developed for the *International Planning Competition* (IPC)[8] of the *International Conference on Artificial Intelligence Planning and Scheduling* (ICAPS). This language is used in the competition to compare the benchmarks of different planning approaches [129].

PDDL evolved to cover the needs of the new IPC competitions[9] :

- PDDL 1.2 [89] (IPC-1998 and IPC-2000) is the original definition of PDDL. It contains STRIPS and ADL functionalities and implements the use of typed variables;
- PDDL 2.1 [46] (IPC-2002) extends the original PDDL to add numeric variables and durative actions;

[7]Not to be confused with Architecture Description Language.
[8]`http://ipc.icaps-conference.org/`
[9]`http://ipc.informatik.uni-freiburg.de/PddlResources`

- PDDL 2.2 [31] (IPC-2004) extends the previous version by adding derived predicates and timed initial predicates;
- PDDL 3.0 [50] (IPC-2006) extends the expressivity of the previous language by adding state trajectory, goal and state trajectory preferences;
- PDDL 3.1 (IPC-2008 and IPC-2011) is the most recent version of PDDL; it introduces *functional* STRIPS [48].

Even if PDDL covers all the previously described functionalities, the majority of planners only implement the STRIPS subset [65].

3.5 The Algorithms

Two main approaches exist to solve classical planning problems [65]: (1) generating a search space and looking for a solution plan in this space; (2) translating the planning problem into a problem that can be solved by a different approach.

3.5.1 Search for Planning

Search algorithms systematically generate and explore a search space looking for a solution path. These algorithms are characterized by the following features:

a) The search space

The search space can be either a state space, a plan space, or a planning graph.

- A node in a *state space* corresponds to a state and an arc corresponds to the execution of an action. Figure 3.3a shows the state space for the blocks-world problem (Figure 3.2). For readability purposes, the actions name where omitted in the arcs. The arcs are bidirectional because there are actions that can *undone* the effect of another actions (*e.g.,* the action that moves the block C from the block A to the block B ($Move(C, A, B)$) can be undone with the action that moves the block C from the block B to the block A ($Move(C, B, A)$)).
- In a *plan space* the nodes are partially specified plans and the arcs correspond to plan refinement operations. Figure 3.3b illustrates a plan space for another block-world problem. The construction of a plan space begins with an empty plan containing a `Start` action with the initial state as an effect ($On(B, D) \wedge On(D, C) \wedge On(C, Table) \wedge On(A, Table)$) and a `Finish` action with the desired goal as a precondition ($On(A, B) \wedge On(C, D)$). Then, a node's predecessor is generated by picking an open precondition (*i.e.,* one predicate of the precondition that is not achieved) and an action to achieve it (*e.g.*, the precondition $On(A, B)$ is achieved with the action $Move(A, Table, B)$). The construction of the plan finishes when there are no more open precondition. That is when all the actions' preconditions are achieved by another action or by the `Start` action. Heuristics can influence the decision of what action is going to be chosen from the set of possible actions. An advantage of this search space is the possibility to make a *partial-order plan.* A partial-order plan is a plan that can place two actions into a plan without specifying which one comes first. For example, the action $Move(A, Table, B)$ and the action $Move(C, Table, D)$

do not have a predefined order and can be executed in parallel or in any possible order, but the Action $MoveToTable(D, C)$ and $Move(C, Table, D)$ have a predefined order and this order must be respected.

- A *planning graph* consists of an oriented graph composed of a sequence of layers that correspond to time steps in the plan. It allows to represent the reachable states after executing a certain number of actions. There are two kinds of layers: *(i) Action layer* A_i, the set of actions whose preconditions hold in the previous *state layer*, and *(ii) State layer* S_i is composed by the union of the predicates of the previous state layer and the predicates of the effects added by the actions of the previous action layer. The first state layer S_0 is the initial state of the problem. Figure 3.3c shows a simple example of a planning graph. The empty squares in the action layer represent the fact that we don't do any change to that specific predicate, and the gray lines represent the conflicts between actions or predicates.

b) The search direction

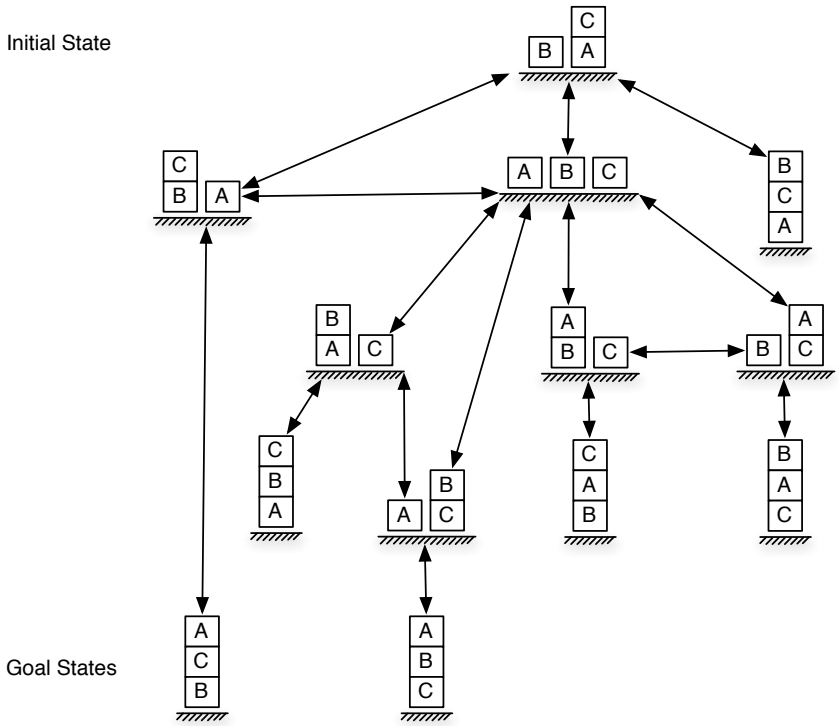
Depending on how the search space is traversed, we can distinguish between forward search, backward search and bidirectional search. The *forward search* starts from the initial node and goes to the goal nodes; the *backward search* starts from the goal node and goes back to the initial node; and the *bidirectional search* executes both searches (forward and backward) until they meet in the middle. The forward search in a state space is known as *progression planning* and the backward search in a state space as *regression planning*.

c) The search algorithms

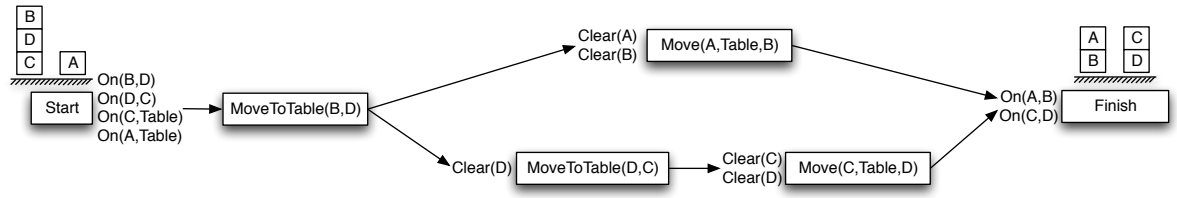
Any search algorithm can be used to find a solution path in a defined search space. An uninformed search strategy [29, 63] systematically traverses the whole search space until finding the solution path (*e.g.,* depth-first search (DFS) [144] and breadth-first search (BFS) [104]). This strategy is not recommended, because the search space in automated planning problems is often extremely large [65]. An informed (heuristic) search strategy [82,111,136] is more suitable for large planning problems because the search algorithm can choose the most promising node, therefore improving the efficiency and scalability of the algorithm. For doing this, a heuristic function $h(n)$ estimates the minimal distance from a node n to the goal. Examples of informed search strategies are: A^* search [58], iterative deepening A^* (IDA^*) [72,73] and recursive best-first search (RBFS) [74,75]. The performance of a heuristic search depends on the quality of the heuristic function.

d) The heuristic function

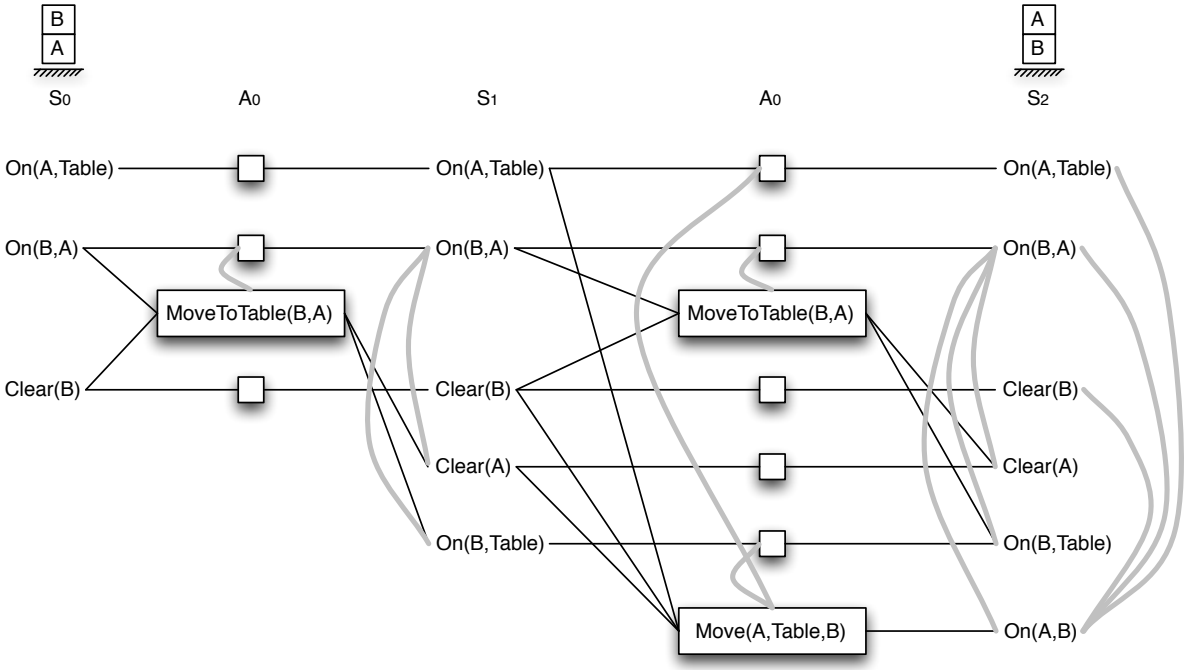
The *heuristic function* $h(n)$ is the minimal estimation of the cost to reach a solution from the node n. The heuristic function can be derived by defining a *relaxed problem* which is easier to solve. For example, considering the following travel problem, we would like to travel from Mons to Paris in the shortest possible way. Instead of calculating the actual highway distance, a possible relaxation is to calculate the straight-line distance (the Manhattan distance) from Mons to Paris. The most common relaxations in automated planning are [129] :



(a) A state space



(b) A plan space



(c) A planning graph

Figure 3.3 – Search spaces for blocks-world problems

Ignore the preconditions : This relaxation ignores all preconditions of actions. Every action becomes applicable in every state. As a consequence, a single predicate of the desired goal can be achieved in one step.

Ignore the delete effect : This relaxation ignores the delete effect of actions (*i.e.*, the negated predicates in the effect). No action will undo the effect made by another action, the solution path of the relaxed problem is a straight and monotonic progress toward the goal. To use this heuristic, the goal and the precondition of the actions need to be composed only by positive predicates.

Subgoal independence : The assumption behind this relaxation is that the sum of the costs of resolving each predicate of the desired goal independently is approximatively the same as the total cost of resolving the complete desired goal.

To have an optimal search algorithm, the heuristic function must be admissible. An admissible heuristic is one that does not overestimate the actual minimal cost of the path. Generally, the heuristic functions implemented in classical planning are non-admissible. In classical planning, the existing admissible heuristics are poorly informed and the application of optimal search algorithms is too expensive in terms of computation time [65].

3.5.2 Planning Solved by a Different Approach

Another strategy to solve classical planning problems is translating the planning problem into a problem that can be solved by a different approach. The most common approaches are :

SAT-planning In this approach all the potential solution plans of a certain length n are translated to a boolean formula. Consequently each assignment of truth values satisfying the boolean formula, represents a valid plan for the planning problem. Given that this approach finds the plans of the given length n, it can be used to find optimal solutions in terms of length of the plan by systematically increasing the value of n [67].

CSP-planning The constraint satisfaction problem (CSP) compilation encodes the potential solution plans of a certain length as CSP problems. Each assignment of values making the resulting CSP problem satisfiable, represents a valid plan for the planning problem. As opposed to the SAT compilation, the CSP compilation is more compact and it can cover planning problems with numeric variables [149].

Planning as model checking In this approach the initial state and the desired goals are formalized (in temporal logic) as requirements about the desired behavior for the plans. The planning problem is solved by searching through the possible plans whether a plan satisfying the requirements exists [21].

3.6 The Implementations

In this section we will present the implementations that influenced significantly the research in automated planning [65]:

STRIPS uses a simple regression planner (backward search in a state-space) [43]. Its representation language was very influential in the creation of future representation languages, such as ADL and PDDL. STRIPS is also considered as the first major planning system.

UCPOP (Universal, Conditional Partial-Order Planning) is a sound and complete algorithm developed for the ADL language [120]. It is based on backward searching in a plan space. This algorithm allows to use conditional effects and universal quantification within the logical conditions and the states. This algorithm is sound, in the sense that all strategies generated by the algorithm are correct solutions for the problem, and complete, in the sense that it always finds a solution for the problem if there is one.

SATPLAN (Planning as satisfiability) [67] translates a planning graph with a determined length into a conjunctive normal form (CNF) formula and solves it with a SAT solver. If no solution is found, the algorithm increases the length of the planning-graph and starts again.

GRAPHPLAN [12] is an algorithm based on a planning graph search space. It alternates between a solution extraction step (that looks whether a plan can be found starting at the end and searching backwards) and a graph expansion step (that adds the actions for the current level and the state predicates for the next level).

FF (Fast-Forward Planning System) is a progression planner (forward search in a state-space) [61, 62]. It is considered by *Russell and Norvig* [129] as the most successful state-space searcher. It was awarded for its outstanding performance at the AIPS 2000 planning competition and top performer at the AIPS 2002 planning competition. FF uses a heuristic function derived from the relaxation of *ignore the delete effect*. The number of actions in the solution of the relaxed problem is used as input to a heuristic algorithm enforced hill-climbing (EHC).

SHOP (Simple Hierarchical Ordered Planner) [108] is a type of Hierarchical Task Network (HTN) planning [131,132,145]. The particularity of this kind of planner is that it decomposes the tasks into subtasks until a task can be performed by a planning action. The planner needs, as input, the information about how the tasks can be decomposed. A task can have more than one possible decomposition. In that case, SHOP performs a forward search to choose which decomposition to apply.

3.7 Automated Planning and Software Engineering

Few other authors use automatic planning to solve different kinds of software engineering problems. For example, Javier Pérez [121] applies, in his PhD thesis, automated planning in order to correct design smells in Java. The author uses JSHOP2 [108] and HTN planning to compute the refactoring strategies needed to automatically correct the design smells. Sirin *et al.* [138,163] use the SHOP2 planner to address the problem of automated composition of web services. Automated planning is also used by Memon *et al.* [91] to automatically generate test cases for graphical user interfaces (GUIs). They use the Interference Progression Planner (IPP) [71] to generate plans representing the testing sequences of GUI interactions.

4

Automated Planning for Inconsistency Resolution

As mentioned in Chapter 1, resolving model inconsistencies is one of the main challenges in *model-driven software engineering (MDE)*. In this dissertation, we propose to use automated planning techniques, presented in Chapter 3, for this purpose. In this chapter, we explain how automated planning can be used for resolving model inconsistencies. We present two different planning approaches: FF and Badger. We study their feasibility in the domain of model inconsistency resolution and make a small scalability study. We conclude this chapter with a discussion about the strengths and limitations of these approaches.

The work presented in this chapter was previously published in:

(i) *"Resolving Model Inconsistencies with Automated Planning"*, presented in 3rd Workshop on Living with Inconsistencies in Software Development, 2010, co-authored with Tom Mens (University of Mons, Belgium) and Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium) [122].

(ii) *"Automated Planning for Resolving Model Inconsistencies - A Scalability Study"*, presented in MoDELS workshop on Models and Evolution, 2010, co-authored with Tom Mens (University of Mons, Belgium) and Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium) [124].

(iii) *"Comparing Automated Planning Approaches for Model Inconsistency Resolution"*, Technical Report, University of Mons, 2011, co-authored with Tom Mens (University of Mons, Belgium) and Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium) [123].

(iv) *"Badger: A regression planner to resolve design model inconsistencies."* in European Conference Modelling Foundations and Applications (ECMFA), 2012, Winner of the ECMFA 2012 Best Foundation Paper Award, co-authored with Tom Mens (University of Mons, Belgium) and Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium) [125].

4.1 Running Example

There is a wide variety of software modeling languages, domain-independent as well as domain-specific. As a consequence, there are many different types of, often interrelated, models that can suffer from many kinds of inconsistencies, such as structural and behavioural inconsistencies.

For our experiments, in this chapter and in the following ones, the Unified Modeling Language (UML) is used to express design models because it is the de-facto general-purpose modelling language [117]. Its visual notation consists of a set of different diagram types, such as class diagrams, sequence diagrams and statecharts, each expressing certain aspects of a software system. These diagrams are interrelated and inconsistencies in and between them can arise easily.

For our running example, we focus on one type of model, namely class diagrams, and on structural model inconsistencies. Figure 4.1 illustrates a simple class diagram containing two inconsistencies of type "Inherited Cyclic Composition" (ICC) and two of type "Cyclic Inheritance" (CI) [151]. An ICC inconsistency occurs when a composition relationship and an inheritance chain form a cycle that would produce an infinite containment of objects upon instantiation. A first inconsistency ICC_1 of this type appears in the inheritance chain `Vehicle` ← `Boat` ← `Amphibious Vehicle`. The second inconsistency ICC_2 occurs in the inheritance chain `Vehicle` ← `Car` ← `Amphibious Vehicle`. Both inconsistencies share the composition relationship between `Vehicle` and `Amphibious Vehicle`.

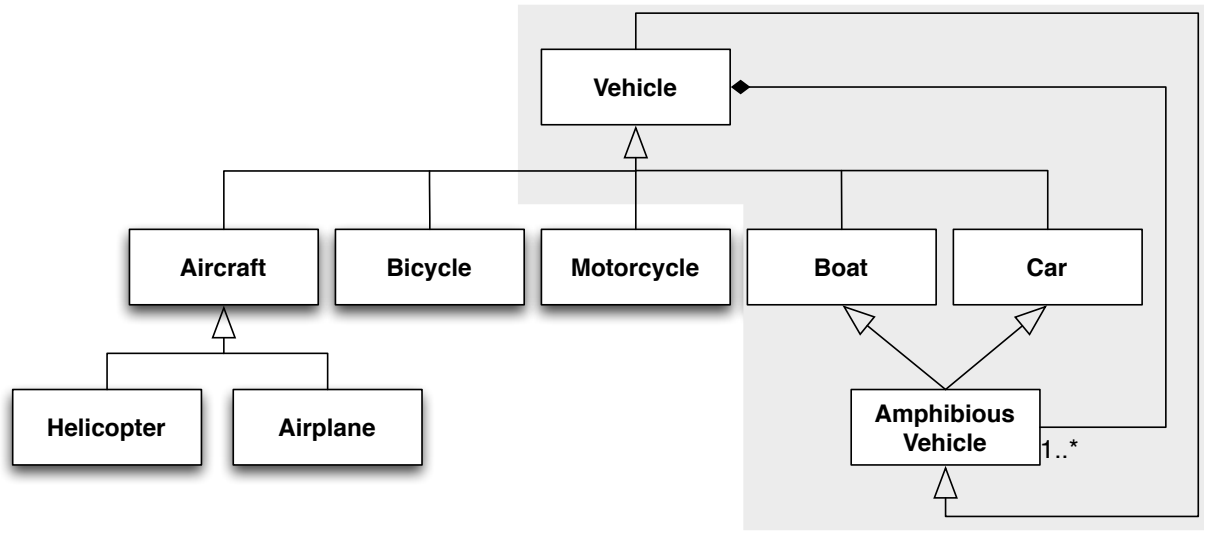


Figure 4.1 – Class diagram with 4 inconsistencies, inspired by [151].

A CI inconsistency arises when an inheritance chain forms a cycle. A first inconsistency CI_1 can be observed in the inheritance cycle involving the classes `Vehicle`, `Boat` and `Amphibious Vehicle`. The second inconsistency CI_2 occurs in the inheritance cycle involving the classes `Vehicle`, `Car` and `Amphibious Vehicle`.

All four inconsistencies share two of the three classes that compose their respective inheritance chains: `Vehicle` and `Amphibious Vehicle`. Due to this overlap, the same resolution action can resolve more than one inconsistency. For example, removing the composition relationship between `Vehicle` and `Amphibious Vehicle` solves the two inconsistencies ICC_1 and ICC_2. Removing the inheritance relationship between `Boat` and `Amphibious Vehicle` solves the two inconsistencies ICC_1 and CI_1. This clearly illus-

trates that, in order to resolve model inconsistencies in an optimal way, it is important to consider all inconsistencies simultaneously.

Figure 4.2 illustrates two different solutions in which the four inconsistencies (ICC_1, ICC_2, CI_1 and C_2) are not present any more. The first solution (top part of Figure 4.2) is obtained after the application of the two following actions: removing the generalization relationship between `Vehicle` and `Amphibious Vehicle` (resolving C_1 and C_2); and changing the multiplicity of the association from `1..*` to `0..*` (resolving ICC_1 and ICC_2). The second solution (bottom part of Figure 4.2) is obtained after the application of the two following actions: removing the generalization relationship between `Vehicle` and `Boat` (resolving ICC_1 and CI_1); and removing the generalization relationship between `Vehicle` and `Car` (resolving ICC_2 and CI_2).

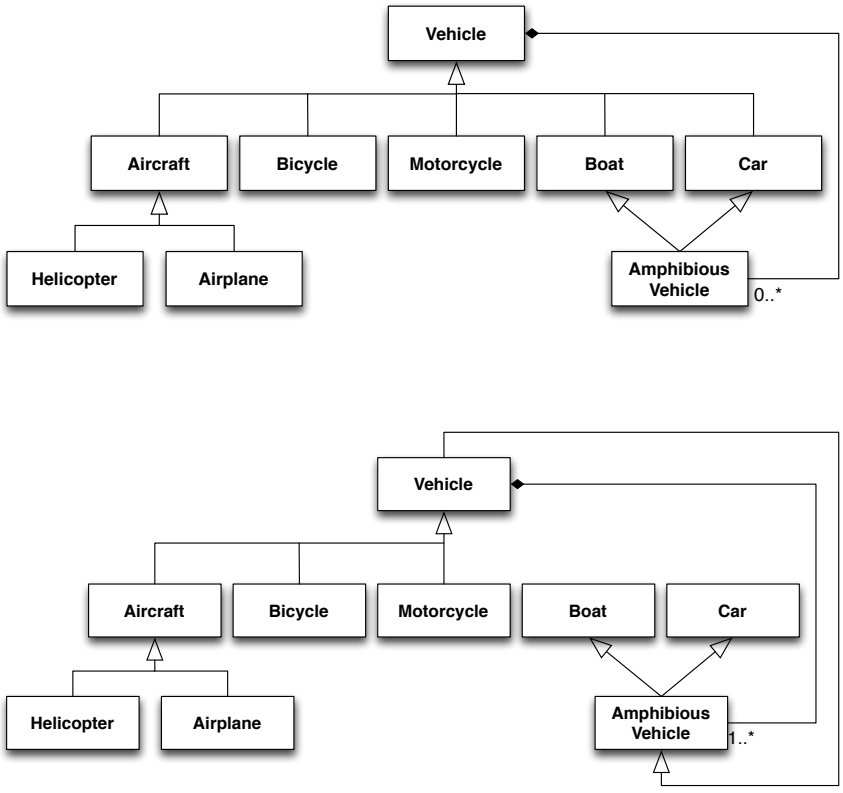


Figure 4.2 – Class diagram without the 4 inconsistencies founded in Figure 4.1.

Note that, in this chapter, we use as running example a very simplified metamodel with a small model and only two types of inconsistencies. In Chapter 6, we will use bigger metamodels, with much bigger models and a large amount of inconsistencty types.

4.2 Planning for Inconsistency Resolution

We have chosen to use automated planning for inconsistency resolution because the solutions proposed by automated planning are discovered and optimised in complex search spaces. Automated planning promise to resolve multiples inconsistencies together in an optimal way (using a minimal number of actions). Automated planning is also a non-intrusive technique, *i.e.*, without the need to change the original problem. In addition, It allows us to generate completely automatic solutions, without the need of user interven-

tion and without the need of adding more hand-coded information to the problem (*e.g.*, resolution rules, generator functions, inconsistency causes).

Using the example of Figure 4.1, we illustrate how to resolve inconsistencies with automated planning. We require as input: an initial state (the inconsistent model), a set of possible actions (that change the model) and a desired goal (the absence of model inconsistencies). Planning requires logic conditions as input, so the whole model environment (*i.e.*, model, meta-model, detection rules) is translated into a conjunction of logic predicates.

The logic predicates used to represent the class diagrams are shown below. A predicate is represented by its name (starting with an uppercase letter) and the arguments that composed it. The arguments are enclosed in parentheses and separated by commas. Logic variables start with a lowercase letter. Literals start with an uppercase letter (*e.g.*, $Parent(Tom, Mathieu)$). The logic predicates used to represent the class diagram are referred to by a unique *id*.

$Class(id, name)$
$Generalisation(id, label, child\_class, parent\_class)$
$Association\_End(id, class, role, upper\_mult, lower\_mult, composite)$
$Association(id, name, ass\_end\_1, ass\_end\_2)$

Initial State. The initial state is expressed as a conjunction of predicates, and represents the current world. In our case, the initial state will be the inconsistent model. The initial state can be represented either by using the *complete model*, or by using a *partial model*, *i.e.*, a model that contains only those elements that are involved in one or more inconsistencies. Using the partial model as initial state, we may not be able to resolve all inconsistencies, or to propose all existing resolutions to the user, which may lead her to resolve the inconsistencies in an undesirable way. But as it is composed only by the elements involved in the inconsistencies, the initial state will be smaller and this could be an advantage for the resolution approach performance.

Below is an example of a *partial model* containing only the elements that are involved in the inconsistencies, shown in the shaded part of Figure 4.1.

$Class(C1, Vehicle) \ \wedge \ Class(C5, Boat) \ \wedge \ Class(C6, Car) \ \wedge$
$Class(C9, Amphibious\_Vehicle) \ \wedge \ Generalisation(G4, Label4, C5, C1) \ \wedge$
$Generalisation(G5, Label5, C6, C1) \ \wedge \ Generalisation(G8, Label8, C9, C5) \ \wedge$
$Generalisation(G9, Label9, C9, C6) \ \wedge \ Generalisation(G10, Label10, C1, C9) \ \wedge$
$Association\_End(AE1, C9, Role1, Star, One, No) \ \wedge$
$Association\_End(AE2, C1, Role2, One, One, Yes) \ \wedge$
$Association(A1, ASS1, AE1, AE2)$

Set of Actions. The set of actions contains the actions that can be performed to change the model. Each action is represented in terms of a *precondition* that must hold before the execution and an *effect* due to the execution of the action. In our approach, inspired by *Blanc et al.* [10], the set of actions corresponds to the elementary operations (basically: create, modify and delete) of the different types of model elements that can be derived from the metamodel. These elementary operations, combined with the logic predicates of the metamodel, allow us to compute the list of all possible actions. As an example, the specification of `modify_Association_Name` action whose purpose is to

modify the association name is given below. The precondition of the action specifies that the association must exist. The effect of the action specifies the changes to the current state. In this case, the association with the old name is removed and the association with the new name is introduced to the current state.

$$\begin{aligned} Action: \;& modify\_Association\_Name(id, name, ass\_end\_1, ass\_end\_2, new\_name) \\ Precondition: \;& Association(id, name, ass\_end\_1, ass\_end\_2) \\ Effect: \;& Association(id, new\_name, ass\_end\_1, ass\_end\_2) \;\wedge \\ & \neg\; Association(id, name, ass\_end\_1, ass\_end\_2) \end{aligned}$$

Be aware that precondition checking stands for different concepts in Automated Planning and in Software Engineering. In Automated Planning *preconditions* are formulated as the set of predicates that have to be present in the current state in order to apply the actions. Additional conditions can also be included in the action specification (*e.g.*, to check a variable type, or to validate certain properties or relationships of the variables involved). How these conditions (from now on we will refer to them as *validation*) are defined depends on each particular planner's algorithms and validation implementations (*cf.* Section 4.4 and 4.5). In our example, a *validation* is needed to verify that the variable *new_name* is correctly initialised as a text and that it is different from the old name.

Desired Goal. The desired goal is a partially specified state, represented as a conjunction of predicates using logic quantification. It specifies the objective that we want to reach, namely the absence of model inconsistencies. To express this objective, we can use two alternatives: (1) the negation of the inconsistencies; or (2) the negation of the inconsistency rules. An inconsistency rule is a conjunction of logic predicates representing a pattern that, if matched in the model, detects inconsistencies.

Below we give an example of the "Inherited Cyclic Composition" rule. Note that $Generalisation^{+}(x, y)$ is the transitive relationship of $Generalisation(\_, \_, x, y)$.

$$\begin{aligned} & Generalisation^{+}(a, b) \;\wedge \\ & Association(a1, ass1, ae1, ae2) \;\wedge \\ & Association\_End(ae1, a, role1, upper1, lower, \mathbf{Yes}) \;\wedge \\ & Association\_End(ae2, b, role2, upper2, \mathbf{One}, composite) \end{aligned}$$

The inconsistency between $Vehicle$ ($C1$) and $AmphibiousVehicle$ ($C9$) that matches this rule is given below.

$$\begin{aligned} & Generalisation^{+}(C1, C9) \;\wedge \\ & Association(A1, ASS1, AE1, AE2) \;\wedge \\ & Association\_End(AE1, C9, Role1, Star, One, No) \;\wedge \\ & Association\_End(AE2, C1, Role2, One, One, Yes) \end{aligned}$$

Using the negation of the inconsistencies as desired goal is only possible if the inconsistencies have already been detected previously. Using the negation of the inconsistency rules has the advantage that it can be used to detect and resolve inconsistencies at the same time, but suffers from scalability problems (see further). In both alternatives, logic negation is used to express the absence of inconsistencies in the resulting model. This implies that we need a planning implementation that allows the use of disjunction and negative predicates in the goal.

Plan. A plan is a sequence of actions that transforms the initial model into a model that satisfies the desired goal (*i.e.*, a consistent model). A plan is generated automatically by the planning algorithm, without relying on any domain-specific information. Moreover, the generated resolution plan does not lead to ill-formed models (*i.e.*, models that do not conform to their metamodel's structure) as long as the metamodel structure is given as a part of the problem specification. But a resolution plan can lead to inconsistent models, relative to other kinds of constraints (*e.g.*, OCL constraints). A complete resolution plan that contains only two actions and solves the four inconsistencies of the running example is given below:

$delete\_Generalisation$:
$Generalisation(G10, Label10, C1, C9)$
$modify\_Association\_End\_Lower\_Multiplicity$:
$from$: $Association\_End(AE1, C9, Role1, Star, One, No)$
to : $Association\_End(AE1, C9, Role1, Star, Zero, No)$

The first action removes the generalisation between $Amphibious\ Vehicle$ and $Vehicle$, removing both cyclic inheritance inconsistencies. The second action modifies the lower multiplicity of the association between $Amphibious\ Vehicle$ and $Vehicle$ from 1 to 0. This action removes both inherited cyclic composition inconsistencies. The model resulting after the execution of the plan is the one in the top part of Figure 4.2.

4.3 Experimental Setup

As the aim of this dissertation is to use Automated Planning to resolve design model inconsistencies, we will select existing implementations and algorithms that fulfill our requirements and we will perform a series of experiments to determine their feasibility and scalability in the inconsistency resolution domain. Note that the aim is not to compare all existing planning implementations and algorithms but to find one that fulfills our requirements.

Our first experiment will consist of assessing whether the use of the selected automated planning approach is at all feasible in the domain of inconsistency resolution. We will start by exploring the impact of different ways to provide input to the planner.

The initial state can be specified by giving a *complete model* or a *partial model*, the latter restricts the search space by containing only those elements that are involved in the inconsistencies (shaded part of Figure 4.1). The desired goal can either contain a negation of the inconsistency rules or a negation of the inconsistencies themselves. In order to assess which of the above four choices is feasible, we will compare the timing results of each considered possibility.

To verify how the automated planning approaches perform on larger models, we will conduct three more experiments in which we will artificially increase the size of the example class diagram of Section 4.2 in order to assess how this affects the time needed to generate a resolution plan.

First, we will artificially augment the size of the motivating example of Figure 4.1 by gradually adding a number of isolated classes to the complete model (from 1 class to 20 classes). Since these classes are unrelated to the inconsistencies that the algorithm needs to resolve, they will never be part of the *partial model* and the algorithm will still be able to find the same resolution plan in the same time if the *partial model* is used as initial

state. However, the time taken to generate a plan using the *complete model* as initial state increases as the model size increases.

Secondly, we will study the timing results for models of increasing size using the *partial model* as initial state. The motivating example of Figure 4.1 contains an inheritance chain of classes that is shared by all the inconsistencies. To assess the effect of an increase of the size of the *partial model* on the time needed to compute a resolution plan, we will artificially augment the size of the model by increasing the length of this inheritance chain. We will do this gradually, by adding between one and eight intermediate superclasses, and computing the timing results for each partial model.

Finally, we will verify whether the number of inconsistencies to be resolved affects the timing results. To achieve this, we will restrict the desired goal to generate resolution plans that resolve only 2 or 3 inconsistencies, without affecting the *partial model* of Figure 4.1. We will not do this for 1 inconsistency only, as it would reduce the size of the partial model, making the results incomparable with what we will find for 2 or 3 inconsistencies. Note that all the resolution plans generated by the automated planning approaches presented in this dissertation resolve the inconsistencies. But the user can still prefer a resolution plan over another to resolve the inconsistencies. Assessing the quality of these resolution plans as perceived by the user will be discussed in Chapter 8.

In order to remove noise, each experiment will be conducted 10 times and the average time will be computed[1]. All experiments will be carried out on a 64-bit Apple MacBook with 2.4 GHz Intel Core 2 Duo processor and 4GB RAM, 2.9GB of which will be available for the experiment.

We will perform a regression analysis to compare the growth of the time results with 5 different parametric regression models: a linear model ($y = a\ x + b$), a quadratic model ($y = a\ x^2 + b\ x + c$), an exponential model ($y = b\ e^{a\ x}$), a logarithmic model ($y = a\ \ln x - b$) and a power model ($y = b\ x^a$). The goodness of fit of each type of model will be verified using the coefficients of determination: the R square (R^2) and the adjusted R square ($\bar{R}^2$) [146]. Its value is always between 0 and 1, and a value close to 1 corresponds to a good fit. The difference between the R^2 and the $\bar{R}^2$ is that the latter takes into account the number of parameters of the model, and is more accurate when we try to compare the goodness of fit of models with different numbers of parameters like it is in our case (the quadratic model has 3 parameters and the other models only have 2 parameters). The formula to compute the R^2 is :

$$R^2 = \frac{\sum_i (f_i - \bar{y})^2}{\sum_i (y_i - \bar{y})^2}$$

where f_i are the regression model values, y_i are the observed values and $\bar{y}$ is the mean of the observed values:

$$\bar{y} = \frac{1}{n} \sum_i^n y_i$$

The formula to compute the $\bar{R}^2$ is:

$$\bar{R}^2 = 1 - (1 - R^2) \frac{n - 1}{n - p - 1}$$

[1]In hindsight, it would have been more appropied to use the minimal time instead of the average time since the noise generated by external factors will never be negative.

where n is the sample size and p the number of parameters in the model.

4.4 Fast-Forward Planning System

Many implementations exist of classical planning. In this dissertation, we first use the heuristic state-space progression planner called *FF* (for "Fast-Forward Planning System" [61, 62]) for our experiments. It is considered by *Russell and Norvig* [129] as the most successful state-space searcher, and was awarded for outstanding performance at the AIPS 2000 planning competition and top performer at the AIPS 2002 planning competition. FF has been chosen not only because of its performance, but also because it uses PDDL[2] language with full ADL[3] subset support, including positive and negative predicates, conjunction and disjunction, negation, typing, and logic quantification in the desired goal. This is crucial to our approach, as explained in the previous section. *FF* is the only readily available planner we have found that properly deals with negation. Therefore, we have selected FF for our first experiments.

4.4.1 Representation Language

The syntax of PDDL is Lisp-like. Each logic predicate is a tuple represented between parentheses. The tuple starts with the name of the predicate (starting with an uppercase letter), followed by literals or by variables. The first time that a variable is defined, it is followed by its domain specific type (separated by a "–"). All objects and variables in PDDL have some type. Note that PDDL doesn't provide built-in support for primitive data types. Therefore, we needed to define `String`, `Boolean` and `Cardinal` as user-defined types. `Cardinal` represents a positive integer (including infinity).

The logic predicates used to represent our class diagram are given below using PDDL syntax. Logic variables start with the "?" character. Literals start with lowercase letters. More information about the PDDL syntax can be found in [49].

```
(Class ?id - class_id   ?name - String)
(Generalisation ?id - g_id   ?label - String
     ?child_class - class_id  ?parent_class - class_id)
(Association_End ?id - ae_id   ?class - class_id   ?role - String
    ?upper_mult - Cardinal   ?lower_mult - Cardinal
    ?composite - Boolean)
(Association ?id - a_id   ?name - String   ?ass_end_1 - ae_id
?ass_end_2 - ae_id)
```

Initial State. A partial model corresponding to the model in Figure 4.1 is given below.

```
(Class c1 Vehicle)
(Class c5 Boat)
(Class c6 Car)
(Class c9 Amphibious_Vehicle)
(Generalisation g4 label4 c5 c1)
(Generalisation g5 label5 c6 c1)
(Generalisation g8 label8 c9 c5)
(Generalisation g9 label9 c9 c6)
```

[2]PDDS stands for Planning Domain Definition Language, *cf.* Section 3.4.4.

[3]ADL stands for Action Description Language, *cf.* Section 3.4.4.

```
(Generalisation g10 label10 c1 c9)
(Association_End ae1 c9 role1 star one no)
(Association_End ae2 c1 role2 one one yes)
(Association a1 ass1 ae1 ae2)
```

Set of Actions. The actions in PDDL are composed of 4 parts, each preceded by a keyword: (i) `:action` describes the name of the action; (ii) `:parameters` describes the parameters of the action (*i.e*, the variables that are going to be used in the action); (iii) `:precondition` lists the preconditions predicates that must exist in the state to apply the action; and (iv) `:effect` gives the predicates to be added and removed from the current action.

As an example, the specification of `modify_Association_Name` action whose purpose is to modify the association name is given below in PDDL syntax.

```
(:action modify_Association_Name
 :parameters (?id - id   ?name - String   ?ass_end_1 - ae_id
              ?ass_end_2 - ae_id   ?new_name - String)
 :precondition (Association ?id ?name ?ass_end_1 ?ass_end_2)
 :effect (when (not (= ?name ?new_name))
               (and (not (Association ?id ?name ?ass_end_1 ?ass_end_2))
                    (Association ?id ?new_name ?ass_end_1 ?ass_end_2)))
)
```

In PDDL, the validation of the action is found as the first part of the effect. For example, the validation `(when (not (= ?name ?new_name))` permits to apply the effect only if the new name is different from the old name.

Desired Goal. The "Inherited Cyclic Composition" (ICC) inconsistency rule using the PDDL syntax is given below. Observe that it only specifies an inheritance chain involving three classes. PDDL syntax does not allow to express transitive closure to make the rule more generic, which is an important practical limitation of PDDL syntax.

```
(exists (?a - class_id   ?b - class_id   ?c - class_id)
 (and
   (exists (?g - g_id   ?Label - g_label)
               (Generalisation ?g ?Label ?c ?a))
   (exists (?g - g_id   ?Label - g_label)
               (Generalisation ?g ?Label ?b ?c))
   (exists (?ae - ae_id   ?role - ae_role
                ?upper - upper_cardinal ?lower - lower_cardinal)
        (Association_End ?ae ?a ?role ?upper ?lower yes))
   (exists (?ae - ae_id   ?role - ae_role
       ?upper - upper_cardinal ?composite - boolean)
        (Association_End ?ae ?b ?role ?upper one_l ?composite))
))
```

One of the two inconsistencies (ICC_1 of Section 4.1) that match the rule is given below.

```
(and
  (exists (?g - g_id ?Label - g_label) (Generalisation ?g ?Label c5 c1))
  (exists (?g - g_id ?Label - g_label) (Generalisation ?g ?Label c9 c5))
  (exists (?ae - ae_id ?role - ae_role ?upper - upper_cardinal ?lower - lower_cardinal)
```

```
          (Association_End ?ae c1 ?role ?upper ?lower yes))
  (exists (?ae - ae_id ?role - ae_role ?upper - upper_cardinal ?composite - boolean)
          (Association_End ?ae c9 ?role ?upper one_l ?composite))
)
```

Plan. A complete resolution plan that solves the four inconsistencies of the motivating example is given below[4].

```
delete_Generalisation :
      (Generalisation g10 label10 c1 c9)
modify_Association_End_Lower_Multiplicity :
               from: (Association_End ae1 c9 role1 star one no)
               to:   (Association_End ae1 c9 role1 star zero no)
```

4.4.2 Algorithm

Fast-Forward (FF) is a forward-chaining heuristic state-space planner implemented in C. The planning algorithm of FF uses an heuristic function derived from the relaxation of "ignore the delete effect" (*cf.* Section 3.5.1). This relaxation ignores the delete effect of actions (*i.e.*, the negated predicates in the effect). No action will undo the effect made by another action, the solution path of the relaxed problem is a straight and monotonic progress toward the goal. The number of actions in the solution of the relaxed problem is used as input to the heuristic algorithm *enforced hill-climbing (EHC)* [61, 62]. EHC is a variation of a breadth-first search which is interrupted each time a successor node s' of node s satisfies $h(s') < h(s)$ retaking up the breadth-first search again from s'. In case EHC search fails, FF performs a best-first search. Moreover, the relaxed plans are used to prune the search space. Usually, the relaxed plan contains only the actions really useful in a state, which permits to restrict the successors of any state to those produced by members of the respective relaxed solution.

4.4.3 Experimental Results

We start our experiments by verifying the feasibility of using FF. We explore the impact of different ways to provide input to the planner, as explained in the experimental setup in Section 4.3. We specify the initial state by giving a *complete model* or a *partial model* containing only those elements that are involved in the inconsistencies (shaded part of Figure 4.1). The desired goal can either contain a negation of the inconsistency detection rules or a negation of the inconsistencies. Table 4.1 summarises the timing results for each combination of choices.

Based on these experiments, we observe that using the negation of the inconsistency rules as desired goal always fails because of memory problems. This is due to the fact that the generated state space becomes too large. The experiments in which the negation of the inconsistencies was used as a desired goal, generated a correct resolution plan. Moreover, the experiment in which a partial model was used as initial state significantly outperformed the experiment where a complete model was used. In both experiments, the resolution plans are the same:

```
delete_Generalisation :
      (Generalisation g10 label10 c1 c9)
```

[4]The syntax of the generated plan does not respect the PDDL syntax for sake of readability.

Table 4.1 – Timing results using FF. Time is expressed in seconds and the standard deviation is mentioned after the ± sign.

Experiment number	Initial state: Model	Desired goal: Negation of	Average time for FF (in seconds)
1	*complete*	*inconsistency rules*	out of memory
2	*partial*	*inconsistency rules*	out of memory
3	*complete*	*inconsistencies*	14.84 ± 0.09
4	*partial*	*inconsistencies*	0.268 ± 0.004

```
modify_Association_End_Lower_Multiplicity :
               from: (Association_End ae1 c9 role1 star one no)
               to:   (Association_End ae1 c9 role1 star zero no)
```

We will now artificially increase the size of the example class diagram of Section 4.2 in order to assess how this affects the time needed to generate a resolution plan.

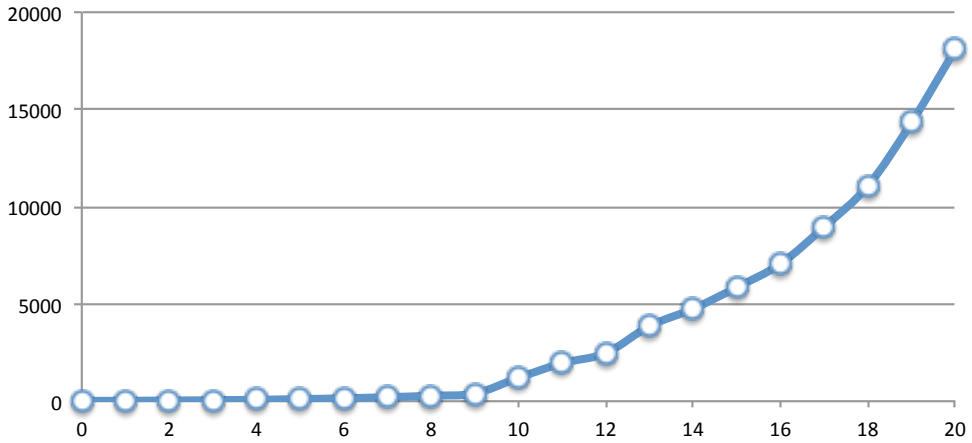
Adding isolated classes to the model. We reran experiment 3 of Table 4.1, while artificially augmenting the size of the model by gradually adding a number of isolated classes. Figure 4.3a illustrates the timing results if we use the *complete model* as initial state. It takes only 15 seconds for our initial example, but it takes more than 5 hours for the model with 20 more added classes. A regression analysis reveals a *quadratic polynomial* growth with adjusted coefficient of determination $\bar{R}^2 = 0.978$, indicating a very good fit of the regression model. Two other candidate regression models we verified had a lower goodness of fit: 0.977 for an *exponential* model and 0.874 for a *power* curve. These results show that FF using a complete model as initial state does not scale up to large models.

In the following experiments, we will only present the results when using a partial model as initial state, since we concluded previously that the use of FF with a complete model does not scale up.

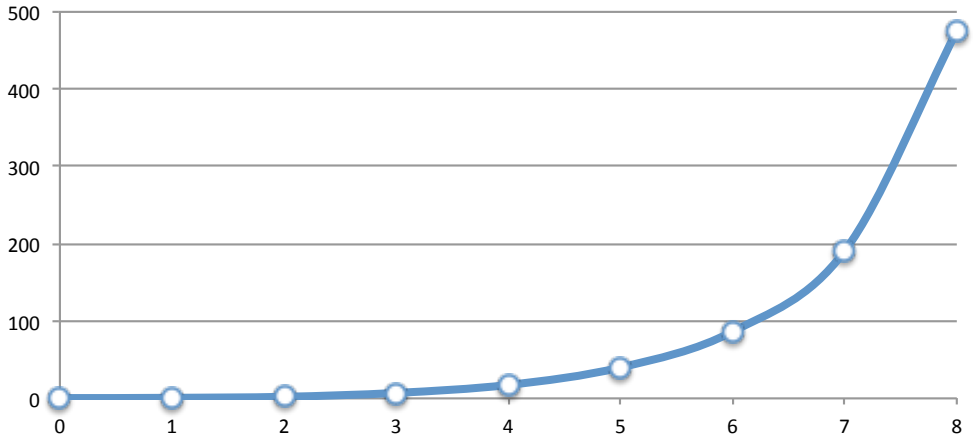
Increasing the inheritance chain of the partial model. We reran experiment 4 of Table 4.1 for models of increasing size. As the introduction of isolated classes does not affect the *partial model* used as initial state, the timing results remain *constant*, irrespective of how many isolated classes are added. To assess the effect of an increase of the size of the *partial model* on the time needed to compute a resolution plan, we artificially augmented the size of the model by gradually increasing the length of the inheritance chains involved in the inconsistencies of Figure 4.1.

Figure 4.3b shows the timing results of carrying out this experiment. The figure shows a strong increase in time to compute the resolution plan as the size of the partial model increases. A regression analysis reveals an *exponential* growth (with adjusted coefficient of determination $\bar{R}^2 = 0.992$) in the time needed to find a resolution plan. Two other regression models we verified had a lower goodness of fit: 0.929 for a power curve and 0.920 for a quadratic polynomial model.

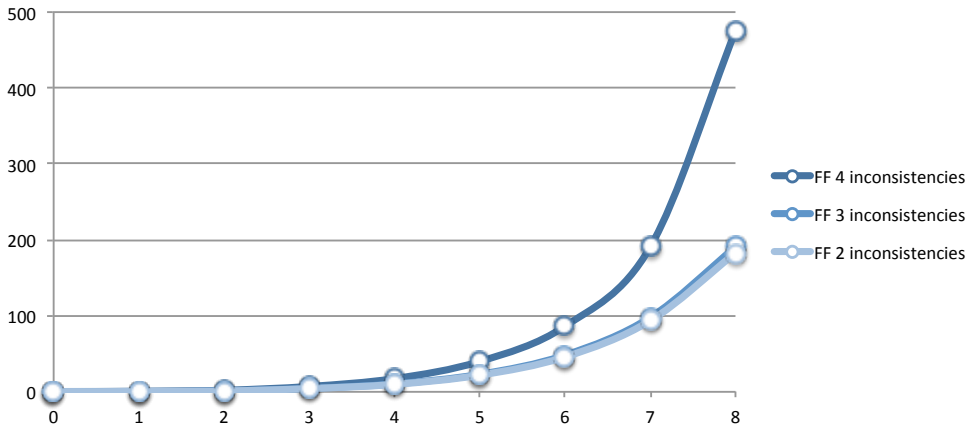
Size of the goal. We verified whether the number of inconsistencies to be resolved affected the timing results. We restricted the desired goal to generate resolution plans



(a) Using the complete model as initial state. The x-axis represents the number of isolated classes added to the initial model.



(b) Adding intermediate superclasses to the partial model. The x-axis represents the number of intermediate superclasses added to the initial model.



(c) For different number of inconsistencies. The x-axis represents the number of intermediate superclasses added to the initial model.

Figure 4.3 – FF - Scalability timing results (the y-axis represents the time in seconds).

that resolve only 2 or 3 inconsistencies, without affecting the partial model of Figure 4.1. We did not do this for 1 inconsistency only, as it would reduce the size of the partial model, making the results incomparable with what we found for 2 or 3 inconsistencies.

In all of these cases we found an *exponential* growth in time (Figure 4.3c). We obtained a goodness of fit $\bar{R}^2 = 0.988$ for resolving 2 inconsistencies, and $\bar{R}^2 = 0.989$ for resolving 3 inconsistencies.

In Figures 4.3b and 4.3c we found the exponential regression model to have the best fit. This was also confirmed by a visual analysis. If we would have more data points (currently we only have 9 values) remains to be seen if the exponential regression model continue to be the best fit.

4.4.4 Discussion

The exponential timing results obtained through the experiments described in the previous section, indicate that using FF to resolve model inconsistencies is not at all usable in practice. Using it to resolve inconsistencies one by one could perhaps be feasible because the partial model and the desired goal will remain relatively small. This is not a good solution, because it does not take full advantage of automated planning. In addition, inconsistencies and their resolution actions are often interdependent. Therefore, resolving the inconsistencies individually may be inappropriate. Another important limitation we encountered is the *expressiveness* of the PDDL syntax. It does not offer important features such as transitive closure, primitive types, numbers. A third limitation of FF is that it only generates a single resolution plan. The resolution of several inconsistencies can give rise to several different resolution plans, *i.e.*, different sequences of resolution actions leading to possibly different consistent models.

Several improvements to this approach can be envisioned. A first improvement is to adapt the planning algorithm so that it generates several resolution plans among which the model designer could choose. The scalability problem could be addressed by implementing a *domain-specific planner* that can be optimized by making it more specific and more performant for the specific problem we want to tackle. In addition, since we are not constrained by the PDDL syntax, this would solve the problems of expressiveness we encountered.

The timing results could be improved by using *regression planning* as opposed to *progression planning* [129], as used by *FF*. Progression planning depends mainly on the size of the initial state and does not exclude irrelevant actions. Regression planning works only with relevant actions and depends mainly on the size of the desired goal. Because of this, the search space will be significantly smaller.

4.5 Badger

After the experience with *FF* we can conclude that we do not only need a planner that properly deals with negation but also a representation language that is more expressive than PDDL. In addition the planner should generate more than one solution plan and should be able to take full advantage of the domain knowledge, the domain specific representation and the techniques adapted to the domain (*i.e.*, a *domain-specific* planner). It is for these reasons that we decided to implement our own planner based on an existing algorithm.

We present here a new planner called *Badger*[5], a *regression planner* that we implemented to address the aforementioned limitations. We have chosen to implement a regression planner, because it depends on the size of the desired goal and works only with relevant actions. A *relevant action* is an action that contributes to the achievement of the goal. The search space of a regression planner will be significantly smaller than the one of a progression planner, as the latter depends mainly on the size of the initial state and does not exclude irrelevant actions.

We implemented the planner algorithm in Prolog, since Prolog's built-in backtracking mechanism allows the planner to easily generate several resolution plans among which the user can choose the most suitable one. The definition of the most suitable one is out of the scope of this dissertation but will be discussed in Chapter 8.

4.5.1 Representation Language

The representation language used by Badger is based on Prolog logic facts and rules.

Prolog[6] is a general purpose logic programming language. Prolog represents relations by means of clauses written as the implication of a head from a body (in prolog syntax `head:- body` in logic syntax: $head \leftarrow body$). Prolog is based on Horn clauses. A Horn clause is a clause that has at most one positive literal. A Horn clause can be written in an implication form as $u \leftarrow (p \wedge q \wedge \ldots \wedge t)$. That means that Prolog can only have one predicate in the head. If the body is composed by at least one predicate it is called a *rule* in Prolog. The conjunction between predicates in the body of the rule is represented by a "," and the disjunction is represented by a ";". Clauses with empty body are called *facts*, and they are always true [24].

The logic predicates used to represent our class diagram are given below using Prolog facts. Logic variables start with an uppercase letter. Logic literals start with a lowercase letter.

```
class(Id, Name).
generalisation(Id, Label, Child_class, Parent_class).
association_end(Id, Class, Role, UpperMult, LowerMult, Composite).
association(Id, Name, Ass_end_1, Ass_end_2).
```

Initial State. The initial state is represented as a list of positive facts that represents the input model. This list represents a logic conjunction. Below is an example of a *partial model* containing only the elements that are involved in the inconsistencies, shown in the shaded part of Figure 4.1.

```
[class(c1, vehicle),
 class(c5, boat),
 class(c6, car),
 class(c9, amphibious_vehicle),
 generalisation(g4, label4, c5, c1),
 generalisation(g5, label5, c6, c1),
 generalisation(g8, label8, c9, c5),
 generalisation(g9, label9, c9, c6),
 generalisation(g10, label10, c1, c9),
 association_end(ae1, c9, dst, star, one, no),
```

[5]The name *Badger* comes from the *honey badger*, an animal that is able to run backwards.

[6]The name Prolog comes from the abbreviation of the french phrase: *"PROgrammation en LOGique"* that means logic programming.

```
 association_end(ae2, c1, src, one, one, yes),
 association(a1, ass1, ae1, ae2)]
```

Set of Actions. The logic rules below specify the possible action `modify_Association_Name`. The *precondition* (the `pre` rule) states that the association must exist before it can be changed. The *validation* (the `can` rule) is used to verify that the new name is correctly typed and that is different from the old name. The *effect* (the `adds` and `deletes` rules) expresses the facts to be respectively added and deleted from the current state to change the name of an association.

```
pre(modify_Association_Name(Id, Name, NewName),
    [association(Id, Name, Ass_end_1, Ass_end_2)]).

can(modify_Association_Name(Id, Name, NewName)):-
    string(NewName),
    NewName \== Name.

adds(modify_name_Association(Id, _Name, NewName),
     [association(Id, NewName, Ass_end_1, Ass_end_2)]).

deletes(modify_name_Association(Id, Name, _NewName),
        [association(Id, Name, Ass_end_1, Ass_end_2)]).
```

Desired Goal. The desired goal is represented as a list of positive or negative facts. The negative facts start with $n(X)$. A disjunction is represented as a list of facts preceded by an "or": $or[X, Y]$. This can be read as X *or* Y. Below we give an example of the "Inherited Cyclic Composition" (ICC) detection rule. It specifies an inheritance chain involving only three classes because the planner syntax does not allow to express transitive closure to make the rule more generic. The planner representation language will be changed in Chapter 5 to allow transitive relationships.

```
[generalisation(G1, Label1, C, A),
 generalisation(G2, Label2, B, C),
 association(A1, Name, AE1, AE2),
 association_end(AE1, B, Role1, Upper1, one, Composite1),
 association_end(AE2, A, Role2, Upper2, Lower2, yes)]
```

One of two inconsistencies (ICC_1 *cf.* Section 4.1) that match this detection rule is given below.

```
[generalisation(g4, label4, c5, c1),
 generalisation(g8, label8, c9, c5),
 association(a1, ass1, ae1, ae2),
 association_end(ae1, c9, role1, star, one, no),
 association_end(ae2, c1, role2, one, one, yes)]
```

Plan. Four complete resolution plans that contain only two actions and that each solve the four inconsistencies of the motivating example are given below:

```
Resolution plan 1 :

1. delete_Generalisation(g5, label5, c6, c1)
2. delete_Generalisation(g4, label4, c5, c1)
```

```
Resolution plan 2 :

1. delete_Generalisation(g4, label4, c5, c1)
2. delete_Generalisation(g5, label5, c6, c1)

Resolution plan 3 :

1. delete_Generalisation(g10, label10, c1, c9)
2. modify_lower_Association_End(ae1, 1, 0)

Resolution plan 4 :

1. modify_lower_Association_End(ae1, 1, 0)
2. delete_Generalisation(g10, label10, c1, c9)
```

To limit the number of plans presented, in the next section, we will explain that Badger avoids presenting those plans that contains the same actions in a different order (*equivalent* plans). For instance, Badger will only present to the user the resolution plans 1 and 3.

4.5.2 Algorithm

The algorithm used by *Badger* is based on the algorithm[7] explained in Bratko [13, p. 432]. Badger uses a *best-first search* algorithm to iteratively generate a state space and search for a solution in that state space by expanding the most promising node. To do this, the algorithm needs 3 functions: a *successor function*, an *evaluation function* and a *solution function*.

The *successor function* generates the child nodes of a particular node, and is used to generate the state space. It strongly depends on the problem to be solved. The *evaluation function* f evaluates the child nodes to find the most promising one. It is defined as the sum of an *heuristic function* h and a *cost function* g: $f(n) = h(n) + g(n)$ where $h(n)$ is the minimal estimation of the cost to reach a solution from the node n, and $g(n)$ is the actual cost of the path to reach n. The *solution function* checks if a particular node is one of the solutions. These 3 functions are independent of the search algorithm, which means that we can use different best-first search algorithms (*e.g.*, A^* search [58], iterative deepening A^* (IDA^*) [72, 73], memory-bounded A*, recursive best-first search (RBFS) [74, 75]).

The *heuristic function* used by *Badger* is a known planner heuristic that *ignores the preconditions*. Every action becomes applicable in every state, and a single fact of the desired goal can be achieved in one step. Remember that the desired goal is a conjunction/disjunction of logic facts that represents one or more negations of inconsistencies. This implies that the heuristic can be defined as the number of unsatisfied facts in the desired goal.

The *solution function* used by *Badger* checks if there is no more unsatisfied fact in the desired goal. In this case, Badger returns the plan if it is not *equivalent* to an already generated plan (*cf.* the plan example in Section 4.5.1). If the user wants, she can ask, using the Prolog backtracking mechanism, for the next plan.

The *cost function* used by *Badger* is the user-specified cost of applying each action. These costs affect the order in which the plans are generated. The user can, for example, give more importance to actions that add and modify model elements than to actions

[7] Bratko named the algorithm: *state-space definition for means-ends planning based on goal regression*

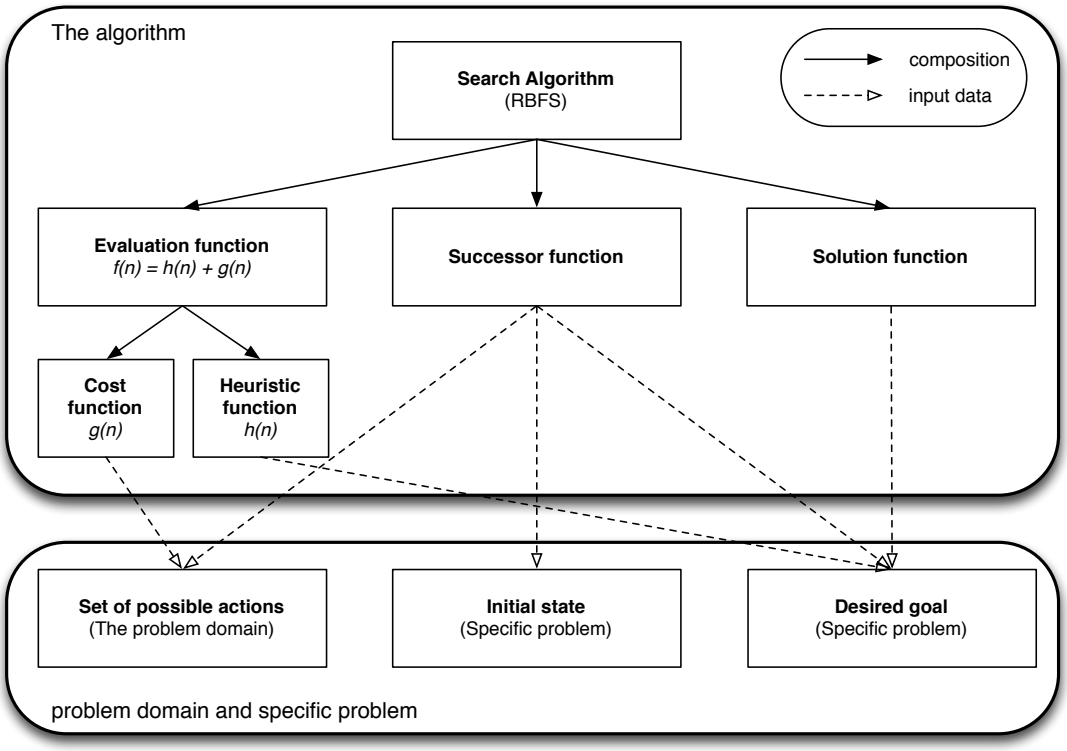


Figure 4.4 – A block diagram representing the functions which compose Badger.

that delete model elements. For the upcoming experiments we use the same cost for each action, implying that the shortest plans are the first generated plans (*cf.* Chapter 7).

The *successor function* is the most complex one and is at the heart of the planning algorithm. It proceeds as follows:

(i) select a logic expression from the desired goal and generate a literal that satisfies this logic expression;

(ii) analyse the effect (the `eff` rule) of each action to find one that achieves this literal;

(iii) validate (the `can` rule) if the selected action can be executed;

(iv) protect the already satisfied literals by checking if the execution of the selected action does not undo a previously satisfied literal;

(v) regress the desired goal through actions by adding the preconditions of the action (the `pre` rule) as new logic expressions in the desired goal and by removing the satisfied logic expression from the desired goal.

Figure 4.4 represents the functions which compose Badger. These functions can be adapted to improve/change Badger:

- any best-first search algorithm can be used as *search algorithm*. We have chosen to use the *recursive best-first search (RBFS)* because it only keeps the current search path and the sibling nodes along this path, making its *space* complexity linear in the depth of the search tree $O(bd)$ where d is the maximum search depth and b is the branching factor. The worst *time* complexity of RBFS is $O(b^{2d-1})$.

- the *successor function* can be changed to improve Badger for inconsistency resolution (*cf.* Chapter 5);

- the *evaluation function* can be changed to adapt the way Badger presents the resolution plans to the users, by changing the order in which the resolution plans are generated (*cf.* Chapter 7);

- and the *solution function* can be changed if the user wants to stop Badger before it resolves all the inconsistencies. For example, the user may want to stop Badger after a certain number of actions, after a certain number of seconds or minutes, after a certain number of inconsistency solved, *etc.* In this dissertation, we aim at resolving all the given inconsistencies and to study the scalability of the approach. For this reason, we will use always the same solution function, one that only stops Badger after resolving all the given inconsistencies, independent of the time or of the number of actions.

4.5.3 Experimental Results

Our first experiment consists of assessing whether the use of Badger is at all feasible. We start by exploring the impact of different ways to provide input to the planner, as explained in the experimental setup in Section 4.3. The initial state can be specified by giving a *complete model* or a *partial model* containing only those elements that are involved in the inconsistencies (shaded part of Figure 4.1). The desired goal can either contain a negation of the inconsistency detection rules or a negation of the inconsistencies.

Table 4.2 – Comparison of timing results using both planners (FF and Badger). Time is expressed in seconds and the standard deviation over 10 different runs is mentioned after the ± sign.

Experiment number	Initial state: Model	Desired goal: Negation of	Average time for FF (in seconds)	Average time for Badger (in seconds)
1	*complete*	*inconsistency rules*	out of memory	N/A
2	*partial*	*inconsistency rules*	out of memory	N/A
3	*complete*	*inconsistencies*	14.84 ± 0.09	0.181 ± 0.003
4	*partial*	*inconsistencies*	0.268 ± 0.004	0.051 ± 0.003

Table 4.2 summarises the timing results for the generation of a single resolution plan for each combination of choices using FF and Badger on the class diagram of Figure 4.1. The negation of the *inconsistency rule* cannot be used as desired goal in Badger because this planner requires all the goals to be completely instantiated, while the negation of *inconsistency rules* is based on variables in the goal. Therefore, the remaining experiments only use the negation of *inconsistencies* as desired goal.

To verify how Badger performs on larger models, we conducted a series of experiments as explained in the experimental setup in Section 4.3, in which we artificially increased the size of the example class diagram of Section 4.2 in order to assess how this affects the time needed to generate a resolution plan. Even if Badger is capable of generating

multiple plans, in the upcoming experiments we only generate a single resolution plan to be able to compare Badger with FF.

Adding isolated classes to the model. First, we reran experiment 3 of Table 4.2, while artificially augmenting the size of the model by gradually adding a number of isolated classes. FF took more than 5 hours for 20 added isolated classes. When repeating the same experiment with Badger, we observe that it outperforms FF with several orders of magnitude (Figure 4.6). For 20 added isolated classes, Badger takes 1.21 seconds.

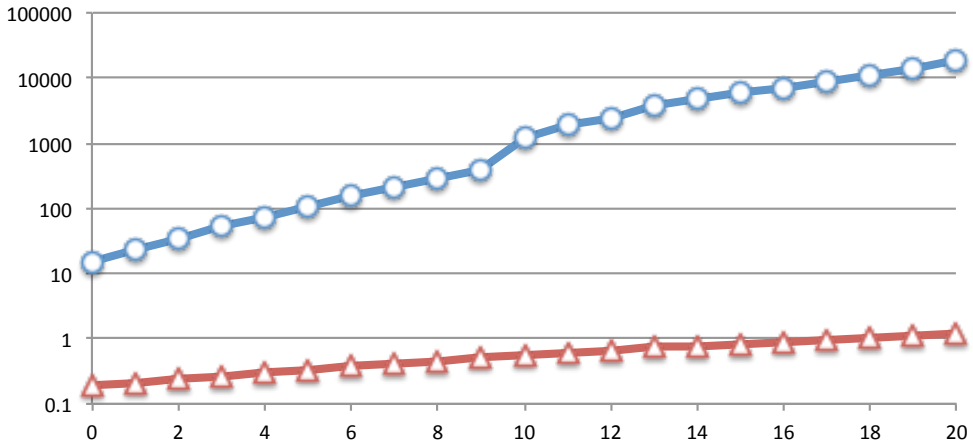


Figure 4.5 – Timing results for FF (blue circles) and Badger (red triangles) using a complete model (experiment 3 of Table 4.2). The y-axis represents the time in seconds on a logarithmic scale. The x-axis represents the number of isolated classes added to the initial model.

We performed a regression analysis to compare the growth with 5 different models: a linear model, a quadratic model, an exponential model, a logarithmic model and a power curve. Based on the results of this regression analysis, shown in Table 4.3, we found that the results of FF grow more rapidly (the quadratic and exponential model offer the best fit) than Badger: still the quadratic and exponential model are the best fit but the parameter a is very small, implying that the growth remains very slow, and close to linear, as can be confirmed by a good $\bar{R}^2$ value for the linear model.

Increasing the inheritance chain to the partial model. We also reran experiment 4 of Table 4.2 for models of increasing size. As the introduction of isolated classes does not affect the *partial model* used as initial state, the timing results remain *constant*, irrespective of how many isolated classes are added. To assess the effect of an increase of the size of the *partial model* on the time needed to compute a resolution plan, we artificially augmented the size of the model by gradually increasing the length of the inheritance chains involved in the inconsistencies of Figure 4.1. Figure 4.6 shows the timing results obtained with Badger and FF, after adding between 1 and 8 intermediate superclasses.

For this experiment, regression analysis reveals that the exponential model is the best one for FF, whereas for Badger the quadratic model is better (though the exponential is also still very good). This was also confirmed by a visual analysis. If we would have more data points (currently we only have 9 values) remains to be seen if these models continue to be the best fit. Table 4.4 gives the detailed results of $\bar{R}^2$ values of all analysed regression models.

Table 4.3 – Comparison of regression models on timing results of Figure 4.5. $\bar{R}^2$ values higher than 0.95 are indicated in **boldface**.

Regression model	$\bar{R}^2$ value for FF	$\bar{R}^2$ value for Badger
linear: $y = a\ x + b$	0.743 ($a = 790,4$)	**0.972** ($a = 0.0493$)
exponential: $y = b\ e^{a\ x}$	**0.977** ($a = 0.3698$)	**0.993** ($a = 0.0939$)
power: $y = b\ x^a$	0.874 ($a = 2.563$)	0.898 ($a = 0.6531$)
quadratic polynomial: $y = a\ x^2 + b\ x + c$	**0.978** ($a = 80.84$)	**0.999** ($a = 0.0015$)
logarithmic: $y = a\ \ln x - b$	0.410 ($a = 4494$)	0.730 ($a = 0.3163$)
Conclusion	Exponential or quadratic growth	Very slow growth, very close to linear

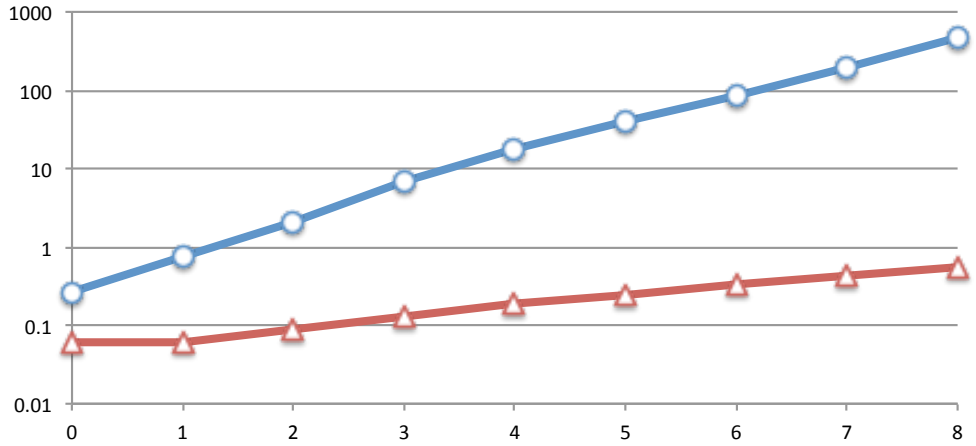


Figure 4.6 – Timing results for adding intermediate superclasses to the partial model using FF (blue circles) and regression Badger (red triangles). The y-axis represents the time in seconds on a logarithmic scale. The x-axis represents the number of intermediate superclasses added to the initial model.

Table 4.4 – Comparison of regression models on timing results of Figure 4.6. $\bar{R}^2$ values higher than 0.95 are indicated in **boldface**.

Regression model	$\bar{R}^2$ value for FF	$\bar{R}^2$ value for Badger
linear: $y = a\ x + b$	0.579 ($a = 22.443$)	0.941 ($a = 0.0701$)
exponential: $y = b\ e^{a\ x}$	**0.992** ($a = 0.1239$)	**0.988** ($a = 0.3168$)
power: $y = b\ x^a$	0.929 ($a = 0.1233$)	0.945 ($a = 1.0849$)
quadratic polynomial: $y = a\ x^2 + b\ x + c$	0.920 ($a = 6.995$)	**0.998** ($a = 0.0074$)
logarithmic: $y = a\ \ln x - b$	0.269 ($a = 63.538$)	0.710 ($a = 0.2206$)
Conclusion	Exponential growth	Quadratic (or exponential) growth

Size of the goal. We verified whether the number of inconsistencies to be resolved affected the timing results. We restricted the desired goal to generate resolution plans that resolve only 2 or 3 inconsistencies, without affecting the partial model of Figure 4.1. As we can see in Figure 4.7, a reduction of the goal that does not affect the size of the partial model does not have a significant impact on the performance. The growth rate and timing results are still similar to what we found in Figure 4.6.

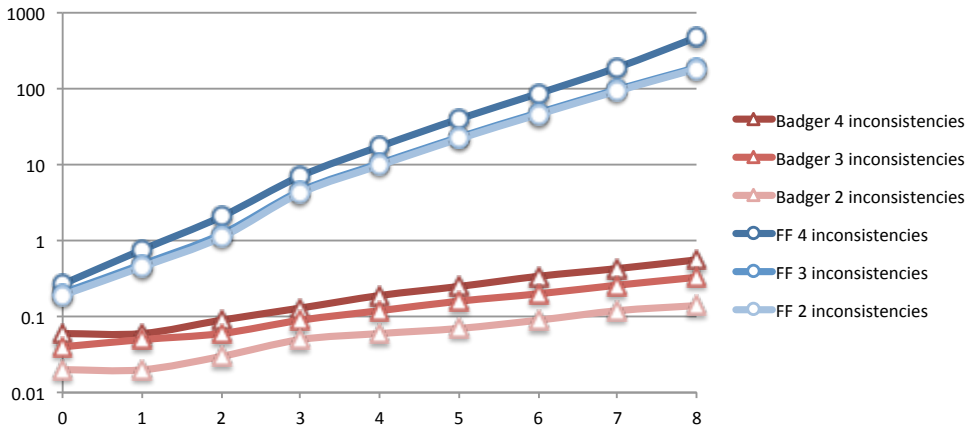


Figure 4.7 – Timing results for progression planning (blue circles) and regression planning (red triangles) for a different number of inconsistencies to be resolved on a partial model. The y-axis represents the time in seconds on a logarithmic scale. The x-axis represents the number of intermediate superclasses added to the initial model.

4.6 Discussion

From all these experiments, we can observe that the type of planner used significantly affects the timing results. The plans generated by both planners were the same in all the performed experiments; even if the representation languages were different in both planners, the initial states, the set of actions and the desired goals were the same, and both planners searched for the shortest plan. While FF does the job, it suffers from scalability and has very poor timing results.

The underlying reason is that FF is optimized for a range of problems in which the search tree is typically narrow and deep, and in which negation is seldomly needed (*i.e.*, typical planning problems like the blocks world problem, the transport problem and the elevators problem[8]). For the purpose of model inconsistencies, we require negation in the desired goal, and we need to deal with search trees that are wide (there are many actions to consider at each step) and shallow (in the worst case the total number of actions in the resolution plan will be proportional to the number of inconsistencies that need to be resolved).

Badger performs significantly better for this type of problem because Badger restricts the search space by excluding many irrelevant actions. Moreover, since we implemented a regression planner ourselves in Prolog, we can still optimise it further to take into account specificities about the problem domain.

The number of actions proposed to resolve an inconsistency involving the modification of a reference in the desired goal depends on the size of the initial state (*i.e.*, it depends on the number of model elements). This negatively affects the performance of both algorithms (Badger and FF) and the number of generated resolution plans (*cf.* Section 5.1).

The representation language that we have used for specifying models was metamodel dependent: for each metamodel element (*e.g.*, `class`), a Prolog predicate was defined, model elements were represented as facts using this predicate (*e.g.*, `class(c1, vehicle)`), and a specific set of actions was defined (*e.g.*, `modify_Class_Name(Id, Name, NewName)`).

The representation language represents the desired goal using only the disjunction or conjunction of positive and negative predicates. The problem with this representation language is that it cannot represent all inconsistencies, *e.g.*, when the inconsistencies use other operators like *greater than* and *less than*. A specific example would be the inconsistency rule checking that the lower multiplicity in an association must be greater than or equal to 0; or the inconsistency rule checking that the higher multiplicity in an association must be greater than the lower multiplicity.

[8] `http://www.plg.inf.uc3m.es/ipc2011-deterministic/DomainsSequential`

5

Badger Improvements for Inconsistency Resolution

In the previous chapter we have seen that even if Badger seems to be promising for inconsistency resolution, some important improvements still need to be done. We explain in this chapter the improvements that we have implemented specifically for inconsistency resolution. We start by presenting the notion of temporary model elements, an abstraction of a set of concrete model elements that allows to reduce the state space search. We also present a metamodel-independent way to represent models. Thanks to this new representation, we can resolve inconsistencies in any kind of structural model. We also introduce new logic operators to describe the desired goal. This improves the number of inconsistency types that Badger can describe. We conclude this chapter with a discussion about the improvements done to Badger.

The work presented in this chapter was previously published in *"Badger: A regression planner to resolve design model inconsistencies."* in European Conference Modelling Foundations and Applications (ECMFA), 2012, Winner of the ECMFA 2012 Best Foundation Paper Award, co-authored with Tom Mens (University of Mons, Belgium) and Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium) [125].

In this chapter, we explain the improvements that we have implemented specifically for inconsistency resolution. The grey part of Figure 5.1 represents the functions that we are going to study and modify, in this chapter, to improve Badger for inconsistency resolution. In Section 5.1 we will change the *set of actions* to include the notion of temporary element. In Section 5.2 we will change the representation language used in the initial state to be metamodel-independent. The *set of actions* will be also changed to correspond to this new representation. In Section 5.3 we will change the *successor function* to allows some new logic operators in the *desired goal*.

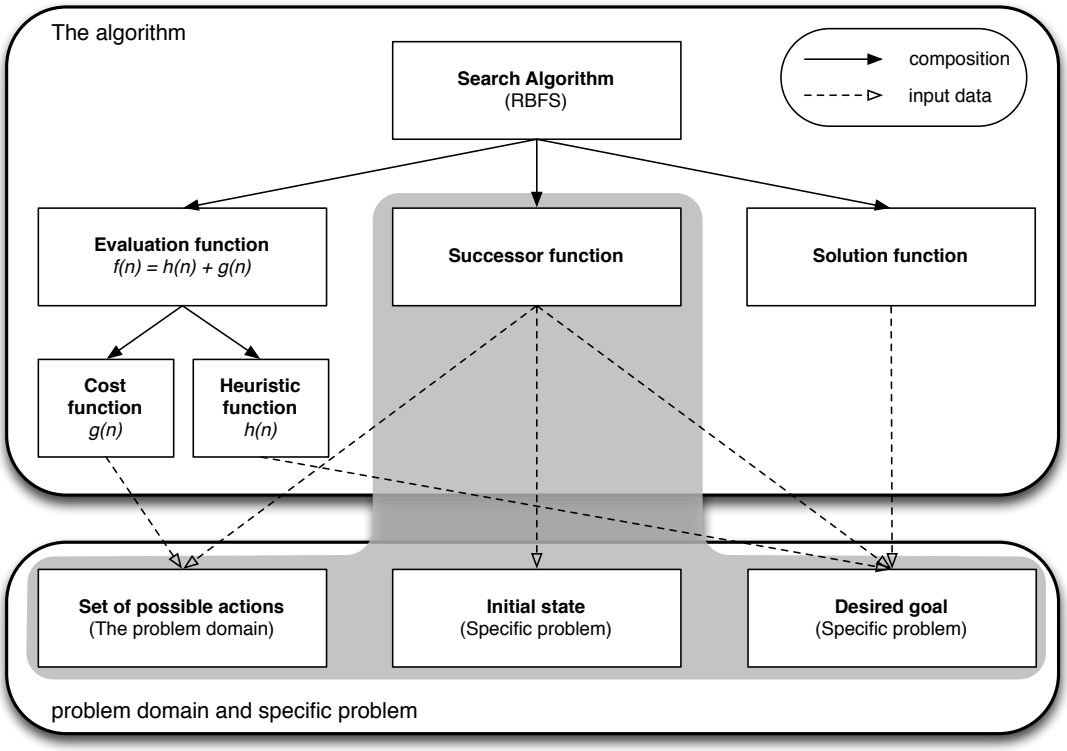


Figure 5.1 – A block diagram representing the functions which compose Badger. The grey part represents the successor function that will be changed in this chapter.

5.1 Temporary Model Elements

As concluded in Section 4.6, the number of actions proposed to resolve a model inconsistency involving the modification of a reference in the desired goal depends on the size of the initial state (*i.e.*, it depends on the number of model elements). This negatively affects the performance of the algorithm and the number of generated resolution plans.

For example, in Figure 5.2, an inconsistency can be observed in the inheritance cycle involving the classes `Car` and `Sports Car`. Badger proposes 22 different solutions[1] to resolve this inconsistency:

1. removing the generalization `g1`;
2. changing the generalization `g1` so that `Car` becomes a subclass of `Seat`;

[1]Note that not all proposed solutions are useful from the point of view of the user, since it depends on the "meaning" of the classes used in the model. For example, conceptually it does not make any sense for a `Car` to be a subclass of `Seat` (or of `Engine`, or `Doors`, *etc.*).

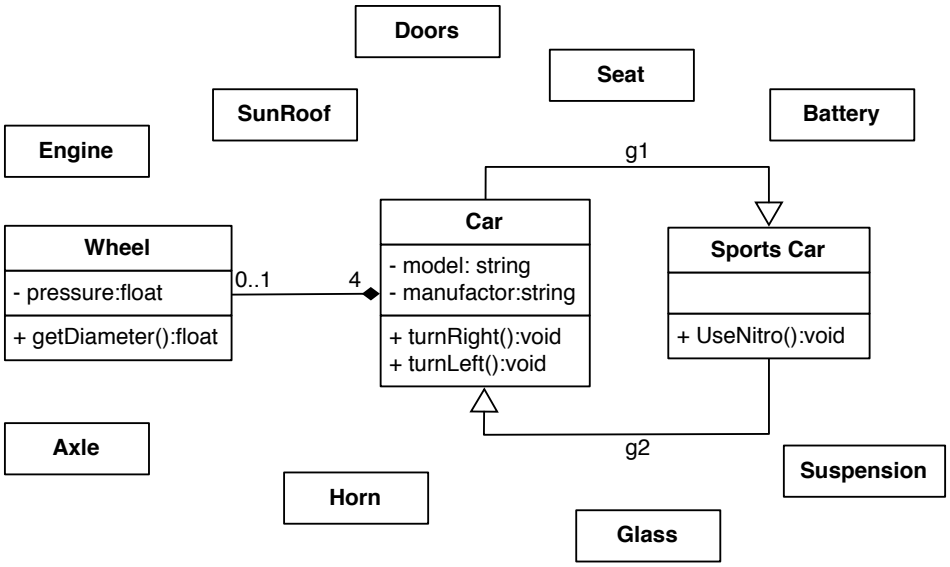


Figure 5.2 – Cyclic Inheritance Examples

3. changing the generalization `g1` so that `Car` becomes a subclass of `Engine`;
4. changing the generalization `g1` so that `Car` becomes a subclass of `Doors`;

⋮

11. changing the generalization `g1` so that `Car` becomes a subclass of `Axle`;
12. removing the generalization `g2`;
13. changing the generalization `g2` so that `Sport Car` becomes a subclass of `Seat`;
14. changing the generalization `g2` so that `Sport Car` becomes a subclass of `Engine`;
15. changing the generalization `g2` so that `Sport Car` becomes a subclass of `Doors`;

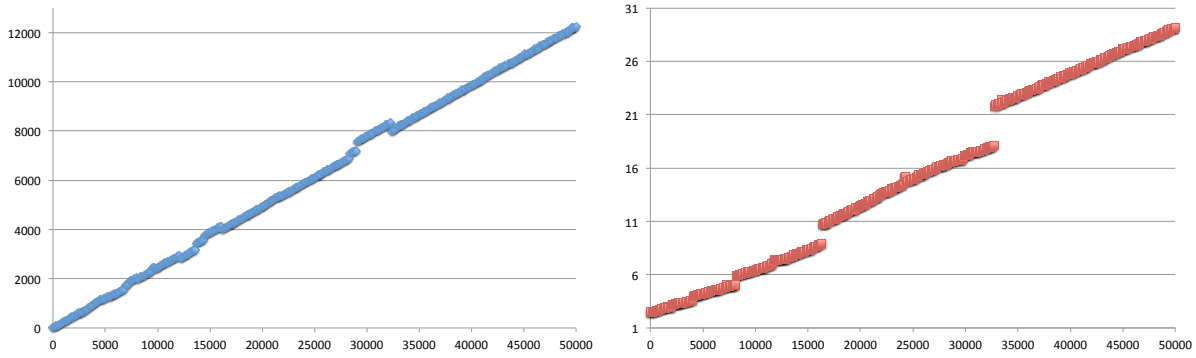
⋮

22. changing the generalization `g2` so that `Sport Car` becomes a subclass of `Axle`.

In this example, the number of solutions depends on the number of classes in the class diagram. To avoid generating many resolution plans, each referring to a concrete model element (*i.e.*, one of the many classes in a class diagram), we introduced the notion of *temporary model element* as an abstraction of such a set of concrete model elements. A temporary model element is represented as a tuple `(+other,X,Y)` where `X` is the model element type (*e.g.*, `class`) and `Y` is the set of model elements of this type that cannot be used as part of the proposed resolution. Once the resolution plan is generated, the user can replace the temporary model element by a concrete model element that does not belong to `Y`, to avoid re-introducing the same inconsistency.

With this notion of temporary model element, Badger will only propose 4 different solutions:

1. removing the generalization `g1`;
2. changing the generalization `g1` so that `Car` becomes a subclass of `(+other,class,['Sports Car'])`;
3. removing the generalization `g2`;
4. changing the generalization `g2` so that `Sport Car` becomes a subclass of `(+other,class,['Car'])`.



(a) Without the notion of temporary model element.

(b) With the notion of temporary model element.

Figure 5.3 – Timing results for resolving one inconsistency with and without the notion of temporary model element. The y-axis represents the time in milliseconds. The x-axis represents the number of isolated classes added to the initial model.

This improves the scalability of Badger, as it proposes only 1 action changing the reference between `Car` and `Sports Car` instead of 10 actions. This also improves the readability of the solutions. It is easier for a user to read that he needs to change a reference to another class than reading a list of solutions in which each solution says that he can change a reference to a precise class. On the other hand, it will require some additional work from the user after the resolution plan has been selected, since the user will need to indicate a concrete model element to replace the temporary model element.

We can illustrate this scalability improvement with a small experiment. We artificially augmented the size of the example of Figure 5.2 by adding an increasing number of isolated classes to the complete model (from 0 to 50000 classes in steps of 100). We compare the time taken by Badger to generate one resolution plan *without* the notion of temporary model element and *with* the notion of temporary model element for all these models.

These experiments were carried on a Power Mac with 2.80GHz Intel Xeon Quad-Core processor and 4Gb RAM. We used the 32-bit version of SWI-Prolog 6.0.2, running on the 32-bit version of Ubuntu 11.04 operating system. All timing results obtained were averaged over 10 different runs to account for performance variations[2]. Figure 5.3 illustrates the timing results without the notion of temporary model element (Figure 5.3a) and with the notion of temporary model element (Figure 5.3b).

	Linear $y = a\ x + b$	Polynomial Quadratic $y = a\ x^2 + b\ x + c$	Exponential $y = b\ e^{a\ x}$	Logarithmic $y = a\ \ln x - b$	Power $y = b\ x^a$
Without the notion of temporary model element	**0.999**	**0.999**	0.726	0.690	**0.990**
With the notion of temporary model element	**0.990**	**0.991**	*0.936*	0.644	0.813

Table 5.1 – $\bar{R}^2$ values of four different parametric regression models used to fit the timing results.

We fitted four different types of parametric regression models with two parameters

[2]In hindsight, it would have been more appropied to use the minimal time instead of the average time since the noise generated by external factors will never be negative.

to the data: a linear model, a logarithmic model, a power model and an exponential model. The goodness of fit of each model was verified using the adjusted coefficient of determination $\bar{R}^2$. Table 5.1 shows the obtained $\bar{R}^2$ values. In order to easily distinguish the best regression models, values higher than 0.90 are indicated in *italics*, while values higher than 0.95 are indicated in **boldface**.

By analyzing the Table 5.1, we observe that the linear and the polynomial quadratic models are the best fit in both cases (whithout and with the notion of temporary model element). In both cases the polynomial models have a growth very close to linear because the a parameter is very close to 0, ($a = 0.00000020314$ and $a = 0.0000000129$ respectively). This explains the excellent results for the linear regression models. A visual interpretation also confirms that the linear models are the best match in both cases.

The time in both cases increases linearly as the size of the model increases but with a very different growth: when Badger does not use the temporary model elements the slope of the linear model ($a = 0.24683$) is 411 times bigger than the slope of the linear model when Badger uses the temporary model elements ($a = 0.00058$). This experiment clearly shows the scalability improvement of the use of temporary model elements.

As discussed in Chapter 4, using partial models as initial state has big performance advantages due to the very small size of the partial model comparing to the complete model. However, when using the partial model as initial state, Badger may not be able to resolve all inconsistencies, or to propose all existing resolutions to the user, which may lead her to resolve the inconsistencies in an undesirable way. This is due to the lack of information in the partial model.

Now, with the notion of temporary model element, Badger is less sensible to the initial state size and allows us to work with the complete model without loosing performance as it is going to be the case for the rest of the experiments in this dissertation.

5.2 Metamodel Independence

As explained in Chapter 1, metamodel independence is crucial to any approach to represent and manipulate different types of models in a uniform way. For specifying models in Badger, we used in Chapter 4 a representation language that was metamodel dependent. For each metamodel element (*e.g.*, `class`), a Prolog predicate was defined, model elements were represented as facts using this predicate (*e.g.*, `class(c1, vehicle)`), and a specific set of actions was defined (*e.g.*, `modify_Class_Name(Id, Name, NewName)`).

In this section we will explain how we changed the representation language of Badger to make it metamodel independent. We will also illustrate the advantage of a metamodel independent tool by using Badger to resolve *problems* in a language different from the modeling language UML. To do this we will show how we have used Badger to propose plans to resolve code smells in a Java program. In Chapters 6 and 7 we have also taken advantage of this new representation to resolve model inconsistencies in different models with different subsets of the UML metamodel without changing the internal representation.

5.2.1 Achieving Metamodel Independence

We changed the way Badger represents models to make it metamodel independent, based on the representation proposed by Praxis [10]. Praxis is a language that represents models and model changes as sequences of elementary model operations.

Elementary model operation	Description
`create(me,mc,r,a)`	create a model element `me` instance of the metaclass `mc`. *e.g.*, `create(c1,class,1,'Jorge').`
`addProperty(me,p,value,r,a)`	add a `value` to the property `p` of the model element `me`. *e.g.*, `addProperty(c1,name,'Vehicle',1,'Jorge').`
`addReference(me1,re,me2,r,a)`	create a reference `re` between two model elements $\texttt{me}_1$ and $\texttt{me}_2$. *e.g.*, `addReference(g10,general,c9,2,'Tom').`
`delNode(me, mc,r,a)`	delete the model element `me` instance of the metaclass `mc`. *e.g.*, `delNode(ae3,property,2,'Tom').`
`remProperty(me,p,value,r,a)`	remove a `value` from the property `p` from the model element `me`. *e.g.*, `remProperty(c1,name,'Vehicles',3,'Jorge').`
`remReference(me1,re,me2,r,a)`	remove the reference `re` between the two model elements $\texttt{me}_1$ and $\texttt{me}_2$. *e.g.*, `remReference(g10,specific,c4,3,'Jorge').`

Table 5.2 – List of elementary model operations and examples of their use for representing class diagrams. The given elementary model operations are performed by the author `a` in a revision `r`.

Table 5.2 lists all elementary model operations that can be used to construct models. For example, the partial model of the class diagram model of Figure 4.1 can be represented as follows in Praxis:

```
1  create(c1,class,1,'Jorge').
2  addProperty(c1,name,'Vehicles',1,'Jorge').
3  create(c5,class,1,'Jorge').
4  addProperty(c5,name,'Boat',1,'Jorge').
5  create(c6,class,1,'Jorge').
6  addProperty(c6,name,'Car',1,'Jorge').
7  create(c9,class,1,'Jorge').
8  addProperty(c9,name,'Amphibious Vehicle',1,'Jorge').
9  create(g4,generalization,1,'Jorge').
10 addReference(g4,general,c1,1,'Jorge').
11 addReference(g4,specific,c5,1,'Jorge').
12 create(g5,generalization,1,'Jorge').
13 addReference(g5,general,c1,1,'Jorge').
14 addReference(g5,specific,c6,1,'Jorge').
15 create(g8,generalization,1,'Jorge').
16 addReference(g8,general,c5,1,'Jorge').
17 addReference(g8,specific,c9,1,'Jorge').
18 create(g9,generalization,1,'Jorge').
19 addReference(g9,general,c6,1,'Jorge').
20 addReference(g9,specific,c9,1,'Jorge').
21 create(g10,generalization,1,'Jorge').
22 addReference(g10,general,c9,1,'Jorge').
23 addReference(g10,specific,c1,1,'Jorge').
24 create(ae1,property,1,'Jorge').
25 addProperty(ae1,name,'src',1,'Jorge').
26 addProperty(ae1,upper,1,1,'Jorge').
27 addProperty(ae1,lower,1,1,'Jorge').
28 addProperty(ae1,iscomposite,'false',1,'Jorge').
29 addReference(ae1,type,c1,1,'Jorge').
30 create(ae2,property,1,'Jorge').
31 addProperty(ae2,name,'dst',1,'Jorge').
32 addProperty(ae2,upper,'*',1,'Jorge').
33 addProperty(ae2,lower,1,1,'Jorge').
34 addProperty(ae2,iscomposite,'true',1,'Jorge').
35 addReference(ae2,type,c9,1,'Jorge').
36 create(assocID,association,1,'Jorge').
37 addProperty(assocID,name,'assocname',1,'Jorge').
38 addReference(assocID,member,ae1,1,'Jorge').
39 addReference(assocID,member,ae2,1,'Jorge').
40 remProperty(c1,name,'Vehicles',2,'Tom').
41 addProperty(c1,name,'Vehicle',2,'Tom').
```

Note that the modification made by Tom and corresponding to revision 2 consists of the two last operations, at lines 40 and 41. This modification changes the name of the class `c1` from `Vehicles` to `Vehicle`.

As shown in Table 5.2, the elementary model operations are metamodel independent, *i.e.*, they can be used for any kind of structural metamodel. The second parameter of each

model operation refers to an element of the metamodel: a meta-class (`mc`) or a meta-class property (`p`) or a relationship between meta-classes (`re`) (*e.g.*, `class`, `ownedattribute`, `parameter`, `ownedoperation`, `name`, `visibility`, `general`, `ownedend`). The two last parameters refers to the author `a` that performed the elementary model operation in a revision `r`.

Besides making the approach metamodel independent, using Praxis for representing models has several more advantages. First, the set of actions is now also metamodel independent. Instead of having rules to create, modify and delete the model elements (*e.g.*, `modify_Class_Name(Id, Name, NewName)`), we now have rules to add, set and remove the elementary model operations (*e.g.*, `setProperty(Id,Property,Value)`). As an example, the logic rules below specify respectively the *precondition* (the `pre` rule), *validation* (the *can* rule) and *effect* (the *eff* rule) of the action `setProperty`.

```
pre(setProperty(Id,MME,Property,OldValue,NewValue),
    [lastAddProperty(Id,Property,OldValue)]).

can(setProperty(Id,MME,Property,OldValue,NewValue)) :-
    mme_property(MME,Property,Type),
    call(Type,NewValue),
    NewValue \== OldValue.

eff(setProperty(Id,Property,OldValue,NewValue),
    [remProperty(Id,Property,OldValue),
     addProperty(Id,Property,NewValue)]).
```

In this case the *precondition* (the `pre` rule) states that the old property must exist before it can be changed. The *validation* (the `can` rule) is used to verify that the new value is correctly typed and different from the old value. This validation is verified by using a *metamodel* that imposes constraints on the model. The *effect* (the `eff` rule) expresses the two Praxis model operations changing the value of a property.

Note, in the example above, the use of the '`last`' prefix in the model operation `lastAddProperty`. This prefix is used to point to the operations in the model that are not followed by other operations canceling their effects. This prefix is used to know the current state of the model. For example, in our model, the *last* operation that assigns a name to the class `c1` is the operation `addProperty(c1,name,'Vehicle',2,'Tom')` in line 41. Contrariwise the operation `addProperty(c1,name,'Vehicles',1,'Jorge')` in line 2 that assigns a name to the class `c1` is not considered as a *last* operation because it is followed by the operation `remProperty(c1,name,'Vehicles',2,'Tom')` in line 40 that cancels the previous operation. For more details, see *Blanc et al.* [10].

The metamodel needed to validate the set of actions (for the class diagram of Figure 4.1), is expressed below as a set of logic facts in Prolog: rule `mme` represents the metamodel elements; `mme_property` represents the properties of the specified metamodel element, the kind of value that is used (*e.g.*, `text`, `boolean`, `int`), the default value and if the property is optional or not; `mme_references` represents the relationships between two metamodel elements, the name of this relationship, the multiplicities and if it is a composite relationship.

```
mme(class).
mme(generalization).
mme(property).
mme(association).
```

```
mme_property(class,name,text,'a class','false').
mme_property(property,name,text,'src','false').
mme_property(property,upper,int,0,'false').
mme_property(property,lower,int,0,'false').
mme_property(property,iscomposite,boolean,'false','false').
mme_property(association,name,text,'assoc','false').

mm_references(generalization,1,1,general,class,0,'*','false').
mm_references(generalization,1,'*',specific,class,0,'*','false').
mm_references(property,1,1,type,class,0,'*','false').
mm_references(association,2,'*',member,property,1,1,'true').
```

The second advantage of using Praxis is that the operation-based representation of models basically uses the same format as the generated resolution plans. As such, applying a resolution plan to a model becomes trivial.

The third advantage is that Praxis comes with a suite of Eclipse plugins: (i) a plugin to reason about ECore and XMI models; (ii) a plugin to reason about the structure of Java and C programs[3]; (iii) a plugin to extract the revisions history of SVN, Git and Mercurial versioning systems[3]; (iv) a peer-to-peer model editing framework [105]; (v) a plugin to generate class diagram models of varying sizes [106] and (vi) an incremental inconsistency detection tool [10].

The model generator will allow us to assess the scalability of our approach, by enabling the generation of useful models of arbitrary large size to be used for our experiments (*cf.*, Chapter 6). As we use the inconsistencies in the desired goal we will only be able to resolve inconsistencies that have already been identified previously. For this detection we can rely on the inconsistency detection approach proposed by *Blanc et al.* [10].

5.2.2 Example of Metamodel Independence : Resolving Code Smells in Java

To illustrate the metamodel independence of Badger, we illustrate how we have used it to resolve three different code smells in a Java program.

Kent Beck[4] defines a *code smell*[5] as a "hint that something has gone wrong somewhere in your code". A more practical definition is that a *code smell* is a pattern that, if detected in the code source, can reveal design problems. A code smell is not an "error" in the source code, it is not going to cause bugs or failures to the program (at least not at the moment, perhaps it can, somewhere in the future). But a code smell can warn about poor quality source code or about design problems that make the system harder to understand, maintain and evolve [6, 121].

Some examples of code smells are: a method with a too large number of parameters (*Long Parameter List* [6, p. 78]); a subclass that does not use some of the methods and fields inherited from its superclass (*Refused Bequest* [6, p. 87]); a code structure that is found in more than one place (*Duplicate code* [6, p. 76]); and a Test Case without tests. A more complete list of examples can be found in Table 5.3.

[3] `http://code.google.com/p/harmony/wiki/VPraxis`

[4] See `http://c2.com/cgi/wiki?CodeSmell`

[5] Some authors use the term *bad smells in code* [6] while other authors use the term *antipatterns* [17], *design defects* (défauts de conceptions) [147], *disharmonies* [80], *design flaws* [148] or *design smells* [87, 121].

For this experiment we chose the Java program FindBugs[6]. FindBugs is an open source program that uses static analysis to identify different types of code smells in Java programs.

We used the Praxis plugin to export the Java program structure into the Praxis elementary model operations. Figure 5.4 illustrates an example of the input and output of the plugin to translate the structure of Java to Praxis elementary model operations. Figure 5.4 represents a small Java program with 3 classes and the right part represents the exported Praxis code. The metamodel of the structure of Java used by Praxis is the one presented in Figure 5.5.

Figure 5.4 – Example of the input and output of the Java to Praxis plugin

Table 5.3 shows the list of code smells detected in FindBugs using the Praxis detection plugin. Note that in addition to this list, other tools may define and detect other code smells. From this list we chose to resolve one code smell from 3 different types : BS_4, BS_8 and BS_{11}. Note that only the structure of the Java programs is exported into Praxis elementary model operations and not the code itself. This restricts us to solve only code smells that affect the structure of the program. For example, a duplicated code is impossible for us to detect and solve, because the code itself its not represented in Praxis.

a) Abstract method overrides a concrete implementation

The first code smell to be resolved using Badger is one of the type BS_4 : "Abstract method overrides a concrete implementation". This code smell arises in FindBugs when the abstract method `sawOpcode` belonging to the class `bcel.OpcodeStackDetector`[7] overrides the concrete method `sawOpcode` belonging to the class `visitclass.DismantleBytecode`. The *extends* chain between both classes is composed of 3 classes:

[6] `http://findbugs.sourceforge.net/`

[7] All the classes involved in the three code smells that we will resolve using Badger lie in the package `edu.umd.cs.findbugs`

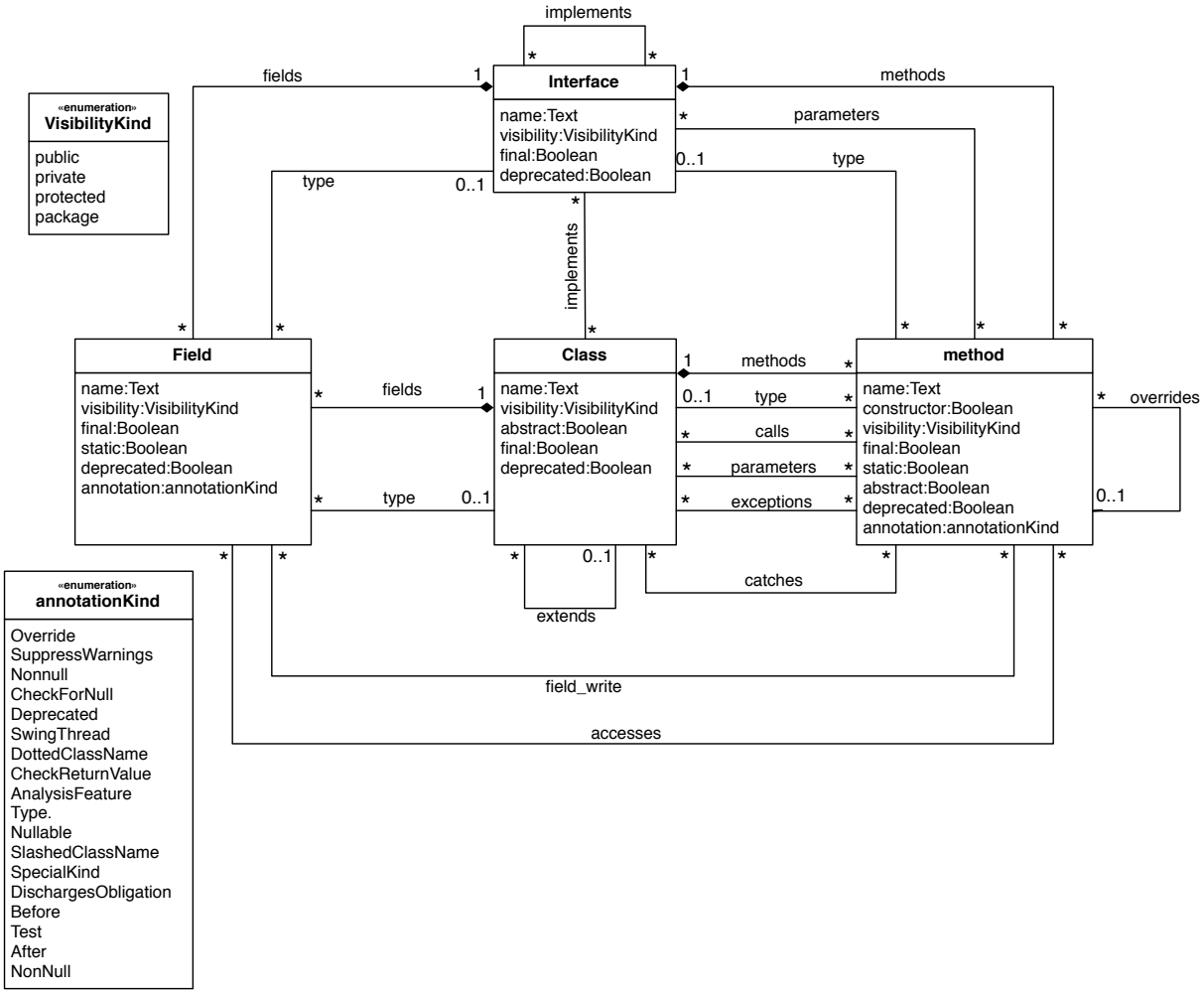


Figure 5.5 – Metamodel of the structure of Java used by Praxis.

Id	Description	Detected code smells
BS_1	TestCase has no tests	1
BS_2	TestCase defines setUp that doesn't call super.setUp()	13
BS_3	Method uses Properties.put instead of Properties.setProperty	1
BS_4	Abstract method overrides a concrete implementation	1
BS_5	Use of class without a hashCode() method in a hashed data structure	102
BS_6	Class relies on internal API classes	1
BS_7	Method creates local variable-based synchronized collection	24
BS_8	Class defines field that masks a superclass field	34
BS_9	Method doesn't override method in superclass due to wrong package for parameter	1
BS_{10}	Constructor makes call to non-final method	437
BS_{11}	Unwritten field	2213

Table 5.3 – List of code smells detected in the Java program FindBugs.

the class `bcel.OpcodeStackDetector` that extends the class `BytecodeScanningDetector` and the latter extends the class `visitclass.DismantleBytecode` (see top of Figure 5.6). To resolve this code smell Badger proposes the following 8 resolution plans :

1. change the name of the method `sawOpcode` of the class `OpcodeStackDetector`;
2. delete the extends reference between `bcel.OpcodeStackDetector` and `BytecodeScanningDetector`;
3. delete the extends reference between `BytecodeScanningDetector` and `visitclass.DismantleBytecode`;
4. change the name of the method `sawOpcode` of the class `DismantleBytecode`;
5. change the method `sawOpcode` from concrete to abstract (in the class `DismantleBytecode`);
6. change the method `sawOpcode` from abstract to concrete (in the class `OpcodeStackDetector`);
7. delete the method `sawOpcode` of the class `OpcodeStackDetector`. This plan has 50 more actions. These 50 actions are to delete all the references from and to this method. No reference should be left if one of the objects that are part of the reference is deleted. These actions have not been written for the sake of readability;
8. delete the method `sawOpcode` of the class `DismantleBytecode`. This planner have also 55 more actions. These 55 actions are to delete all the references from and to this method. No reference should be left if one of the objects that are part of the reference is deleted. These actions have not been written for the sake of readability.

Figure 5.6 graphically shows the code smell and the 8 resolution plans proposed by Badger. The green annotation `<< add >>` represents the objects, properties or references to be added; the blue annotation `<< modify >>` represents the modifications to be made to a property or a reference; and the red annotation `<< delete >>` represents the objects, properties or references to be deleted.

b) Class defines field that masks a superclass field

The second code smell to be resolved is one of the code smell BS_8 detected : "Class defines field that masks a superclass field". This code smell arises from the field `serialVersionUID` from the class `gui2.FilterMatcher` and the field `serialVersionUID` from the class `StackedFilterMatcher`. The latter one extends the class `gui2.FilterMatcher` (see top of Figure 5.7). To resolve this code smell Badger proposes the following 5 resolutions plans:

1. change the name of the field `serialVersionUID` of the class `StackedFilterMatcher`;
2. delete the `extends` reference between both classes;
3. change the name of the field `serialVersionUID` of the class `FilterMatcher`;
4. delete the field `serialVersionUID` of the class `StackedFilterMatcher` and delete the reference between the class and the field;
5. delete the field `serialVersionUID` of the class `FilterMatcher` and delete the reference between the class and the field.

Figure 5.7 graphically shows the code smell and the 5 resolution plans proposed by Badger.

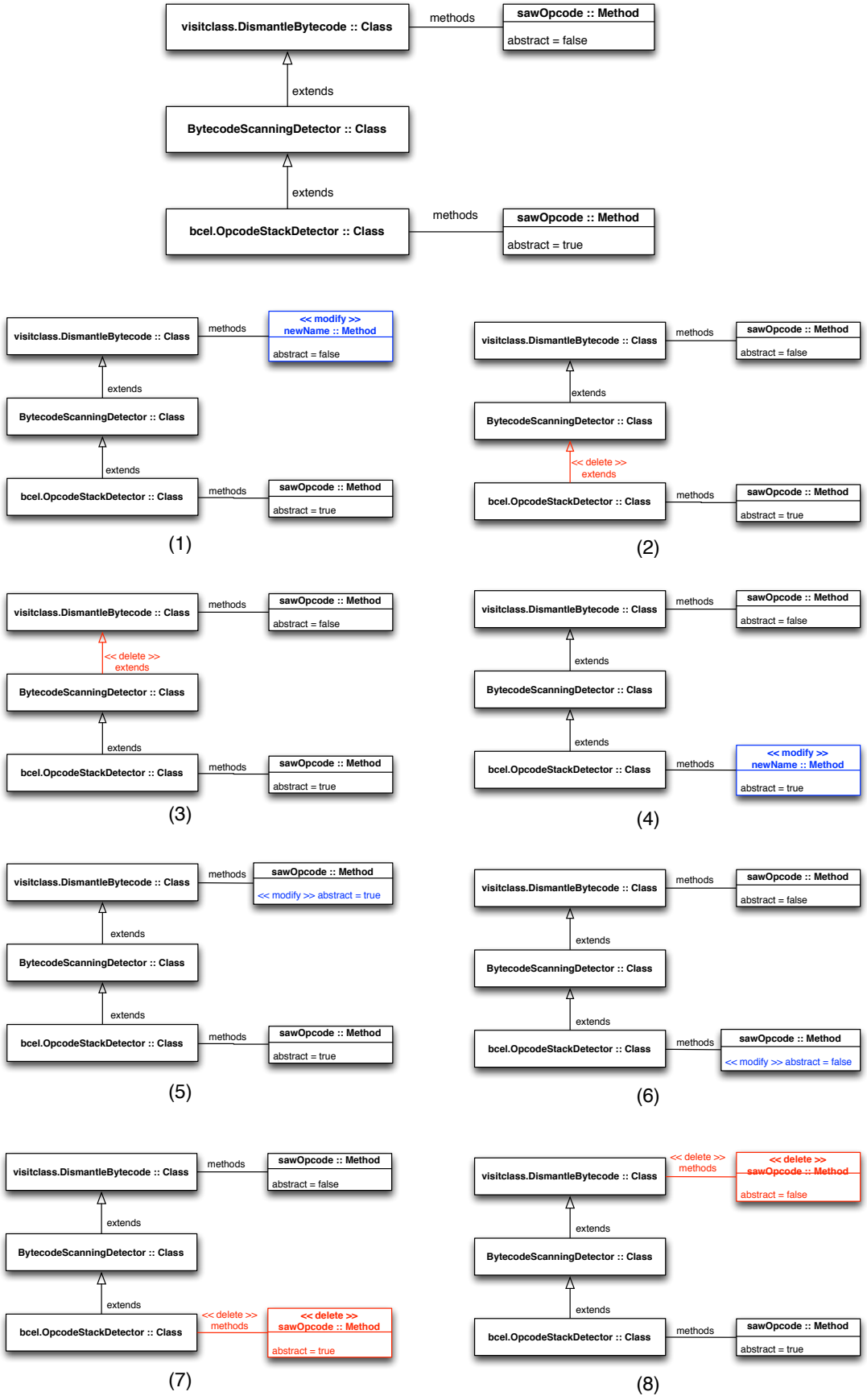


Figure 5.6 – The code smell: "abstract method overrides a concrete implementation" and the proposed resolution plans

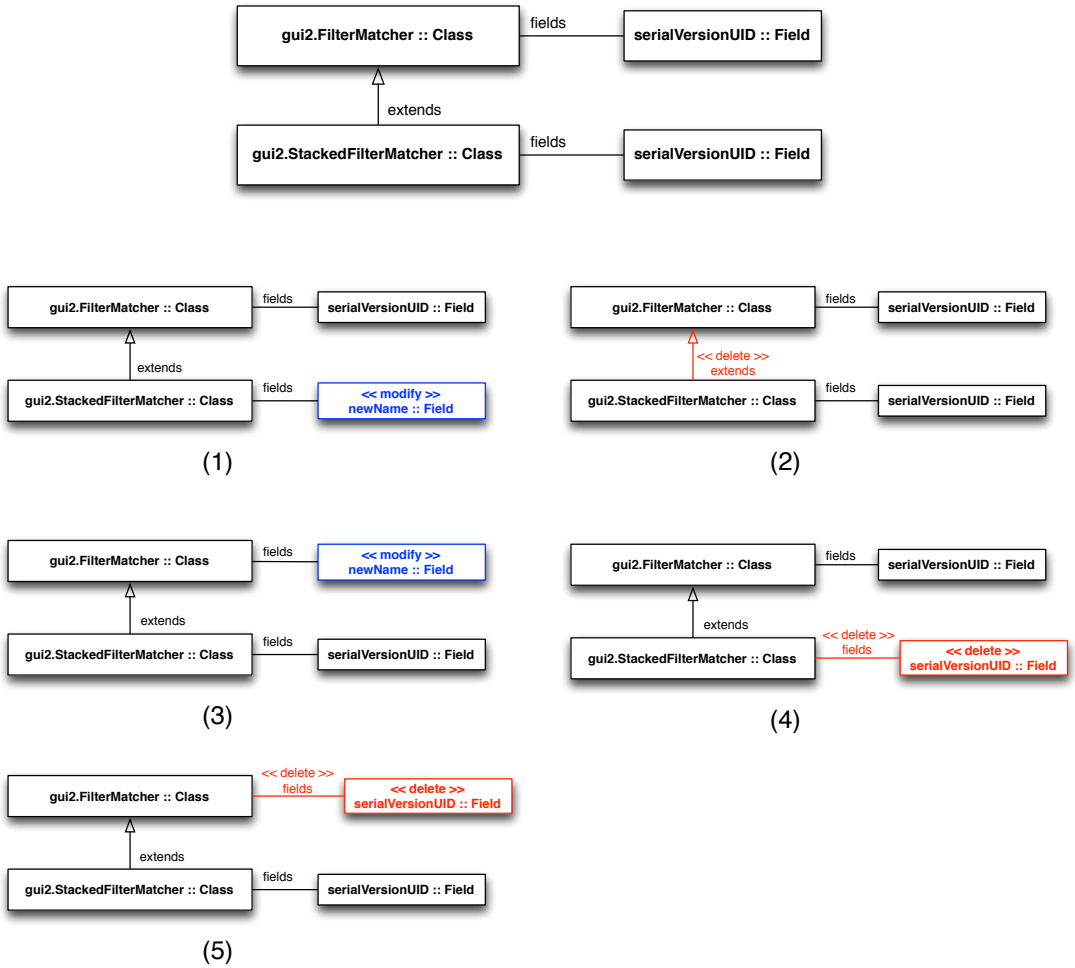


Figure 5.7 – The code smell: "class defines field that masks a superclass field" and the proposed resolution plans

c) Unwritten field

The last code smell to be resolved is one of the type BS_{11}: "Unwritten field". This code smell arises from the field `FINDBUGS_JAR` from the class `anttask.AbstractFindBugsTask` that has not been written by any method (see top of Figure 5.8). The 2 resolutions plans proposed by Badger to resolve the code smell are :

1. add a reference `field_writer` from any method to the field `FINDBUGS_JAR`;
2. delete the field and delete the reference fields from the class to the field.

Figure 5.8 illustrates the code smell and the 2 resolution plans proposed by Badger.

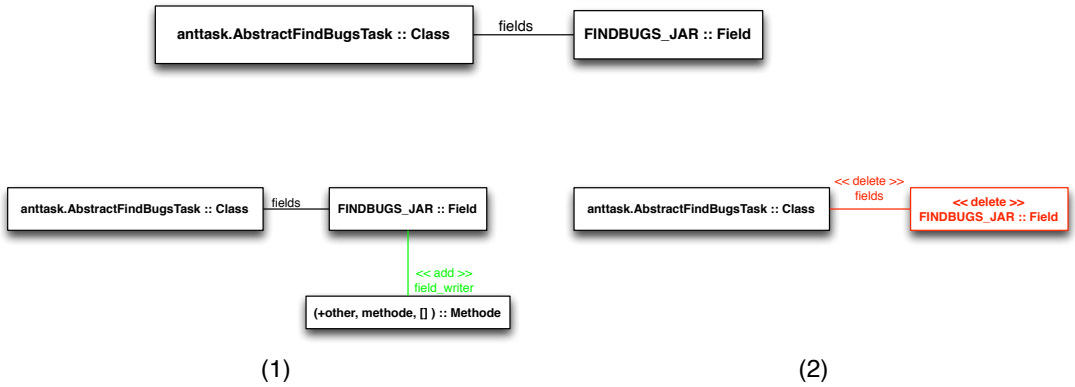


Figure 5.8 – The code smell: "Unwritten field" and the proposed resolution plans.

This experiment with Java, in which we resolve 3 code smells, aims to demonstrate that, thanks to Praxis, Badger is completely metamodel independent. Nevertheless it should be noticed that Badger does not take behavior preservation into account. Even if all the generated plans resolve the 3 code smells, some of the plans change the behavior of the Java program. Normally, when you try to resolve code smells you do not want to change the behavior of the program. For this reason, it is common to apply refactorings to resolve code smells. Refactorings restructure the program code, resolving the code smells, preserving the behavior of the program. A planner algorithm was used by Javier Pérez [121] to compute, from refactoring strategies, the refactoring plans needed to automatically correct code smells. More information about code smells and the strategies to resolve them can be found in his PhD thesis [121].

5.3 Logic Operators

To represent a desired goal in Badger, we only used the disjunction and the conjunction of positive and negative literals (*cf.*, Chapter 4). The problem with this representation is that it cannot represent all inconsistencies, *e.g.*, when they use other operators like *greater than* and *less than* (*e.g.*, the lower multiplicity in an association must be greater than or equal to 0; the higher multiplicity in an association must be greater than the lower multiplicity).

We improved the representation language by adding new logic operators: universal quantification, existential quantification, value comparison, property comparison, counting and transitive navigability. Table 5.4 and Table 5.5 present all logic operators that are allowed to specify the desired goal, inspired by the list of common constructs found

in inconsistency rules [34, 110, 151]. These logic operators will be used for the rest of the experiments in this dissertation. For example, the inconsistencies I_1, I_3, I_4, I_5, I_6, I_8 and I_{10} from the Table 6.1 would be impossible to represent without these logic operators.

Badger does the strict minimum to accomplish this goal. For example, if the user wants to solve the inconsistency "the lower multiplicity must be greater than 0", Badger will propose 1 as solution to avoid an infinite number of possibilities. In the case of two inconsistencies, Badger will also do the strict minimum. For example, in the inconsistencies "the upper multiplicity must be greater than 1" and "the upper multiplicity must be greater than the lower multiplicity", when the lower multiplicity is 3, the upper multiplicity proposed by Badger is 4, as this solution accomplishes both requirements. The same is done with the text values. Badger generates a predefined text (*e.g.*, "Text1") that needs to be changed afterwards by the user. This is also done to avoid an infinite number of possibilities.

Let us illustrate by means of a concrete example how Badger resolves inconsistencies in presence of the logic operators of Table 5.4 and Table 5.5. Suppose we have a cyclic inheritance inconsistency between 3 classes (`a`, `b`, and `c`) as shown in Figure 5.9.

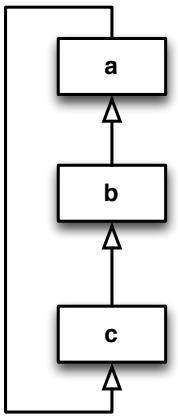


Figure 5.9 – Small class diagram containing a cyclic inheritance inconsistency

The desired goal of this problem is the negation of the inconsistency found in Figure 5.9. This negation can be expressed using our logic operators as: `[not(nav(a,superClass,a))]` and Badger will try to reach this desired goal by executing the following steps:

i. Selecting a logic expression from the desired goal. In our case we only have one:

```
not(nav(a,superClass,a))
```

Starting from this expression, Badger generates a list of literals that satisfy this operator:

```
[not(lastAddReference(a,superClass,b)),
not(lastAddReference(b,superClass,c)),
not(lastAddReference(c,superClass,a))]
```

From this list Badger chooses one literal, for example:

```
not(lastAddReference(a,superClass,b))
```

ii. Analyse the effect of each action to find an action that achieves the literal chosen in step *(i)*. For example, the action `deleteReference(a,MME1,superClass,b,MME2)` can achieve the literal `not(lastAddReference(a,superClass,b))`.

iii. Validate if the action (`deleteReference(a,MME1,superClass,b,MME2)`) can be executed. To achieve this, Badger queries the model to know if `lastAddReference(a,superClass,b)` is present in the model and also to identify what kind of metamodel element `a` and `b` conform to. In our example, this is `MME1 = class` and `MME2 = class`.

iv. In order to protect the previously satisfied literals, Badger tests if the selected actions undo one of these literals. In our example, there are no previously satisfied literals, so no protection is needed.

v. Regress the desired goal through the action `deleteReference` by adding the preconditions of the action to the goal. Badger interrogates the metamodel to know if deleting the reference does not break any metamodel constraint. If a constraint is broken, Badger will generate the literals needed as preconditions. In our example there are no preconditions to add, because deleting the reference does not break any constraint. Hence, the desired goal stays the same:

```
[not(nav(a,superClass,a))]
```

Badger will now remove all satisfied logic expressions from the desired goal. In our example, as the literal `lastAddReference(a,superClass,b)` is not present any more in the model, the logic expression `not(nav(a,superClass,a))` is satisfied and the new goal is an empty goal `[ ]`.

Once the empty goal has been reached, Badger concludes that a goal state is found. The resolution plan will therefore be: `deleteReference(a,class,superClass,b,class)`

If the user ask for the next resolution plan, Badger will rely on the Prolog's backtracking mechanism to choose a different action in step *ii* or a different literal in step *i*. In this way, all possible resolution plans will be generated.

5.4 Discussion

In this chapter we have presented some important improvements implemented specifically for inconsistency resolution in Badger.

We introduced the notion of temporary model elements, an abstraction of a set of concrete model elements that allows to reduce the state space search. This improves the scalability of Badger, as it reduces the state space search, and the readability of the solutions, as it reduces the number of generated plans. On the other hand, the intervention of the user is from now on mandatory, since she will need to indicate a concrete model element to replace the temporary model element.

We presented a metamodel-independent way to represent models using the representation language Praxis. Praxis represents a model as a sequence of elementary model operations. Thanks to this new representation, we can resolve inconsistencies in any kind of structural model. We illustrated the advantage of a metamodel independent tool by using Badger to propose plans to resolve code smells in a Java program.

Atoms

Name		
Praxis primitive	Syntax	`lastCreate(id,mme)` `lastAddProperty(id,property,value)` `lastAddReference(id1,reference,id2)`
	Semantics	The semantics of the *Praxis primitives* is defined by their semantics given by the Praxis language (*e.g.*, `lastCreate(id,mme)` is true if there is a `create(id,mme,r,a)` in the sequence of model operations that is not deleted later in the sequence)
	Example	`lastAddProperty(c1, name,'Vehicle')`
Universal quantification	Syntax	$\texttt{forall}(\texttt{P}(\texttt{x}_1,...,\texttt{x}_\texttt{k},\texttt{y}_1,...,\texttt{y}_\texttt{p}),\texttt{Q}(\texttt{y}_1,...,\texttt{y}_\texttt{p},\texttt{z}_1,...,\texttt{z}_\texttt{m}))$ where P and Q are *Praxis primitives*
	Semantics	$\forall x_1...\forall x_k\forall y_1...\forall y_p(P(x_1,...,x_k,y_1,...,y_p) \Rightarrow \exists z_1...\exists z_m Q(y_1,...,y_p,z_1,...,z_m))$
	Example	`forall(lastCreate(Y,class), lastAddProperty(Y,name,Z))`
Existential quantification	Syntax	$\texttt{exists}(\texttt{P}(\texttt{x}_1,...,\texttt{x}_\texttt{k}))$ where P is a *Praxis primitive*
	Semantics	$\exists x_1\exists x_2...\exists x_k(P(x_1,...,x_k))$
	Example	`exists(lastCreate(X,class))`
Value comparison	Syntax	$\texttt{compare}(\texttt{P}(\texttt{x}),>,\texttt{v})$ where P is a *Praxis primitive* of kind `lastAddProperty`, x is a variable that represents the third parameter of P and $v \in \mathbb{N}$
	Semantics	Let n be the unique value such that the model M satisfies $P(n)$. The truth value of $\texttt{compare}(\texttt{P}(\texttt{x}),>,\texttt{v})$ is the truth value of $n > v$
	Example	`compare(lastAddProperty(ae1,lower_mult,X),>,0)`
Property comparison	Syntax	$\texttt{compare}(\texttt{P}(\texttt{x}),>,\texttt{Q}(\texttt{y}))$ where P and Q are *Praxis primitives* of kind `lastAddProperty`, x and y are variables that represent the third parameter of P and Q
	Semantics	Let n and m be the unique values such that the model M satisfies $P(n)$ and $Q(m)$. The truth value of $\texttt{compare}(\texttt{P}(\texttt{x}),>,\texttt{Q}(\texttt{y}))$ is the truth value of $n > m$
	Example	`compare(lastAddProperty(ae1,upper_mult,X),>, lastAddProperty(ae1,lower_mult,Y))`
Counting	Syntax	$\texttt{count}(\texttt{P}(\texttt{x}),>,\texttt{v})$ where P is a *Praxis primitive*, x is a variable and $v \in \mathbb{N}$
	Semantics	$\lvert\{a \lvert M \models P(a)\}\rvert > v$ where M is the model
	Example	$\texttt{count}(\texttt{lastAddReference}(\texttt{assID},\texttt{member},\texttt{X}),>,2)$
Transitive Navigability	Syntax	`nav(From, Kind, To)` where $From$ and To are model elements and $Kind$ is a binary relation between model elements.
	Semantics	`nav(From, Kind, To)` is true if $(From, To)$ is in the transitive closure of $Kind$.
	Example	`nav(c1,generalization,c9)`

Table 5.4 – Logic Operators - Atoms. Although the operators value comparison, property comparison and counting are only shown with the $>$ function, the other comparison functions can be used as well : $<$, $\geq$, $\leq$, $=$, $\neq$

Boolean Combinations

Name		
Negative literal	Syntax	`not(L)` where L is an *Atom*
	Semantics	not P
	Example	`not(lastAddProperty(ae2,iscomposite,'true'))`
Conjunction	Syntax	$[\mathtt{L}_1, \mathtt{L}_2, ..., \mathtt{L}_k]$ where $L_1, L_2, ..., L_k$ are *Atoms* or *Boolean Combinations*
	Semantics	L_1 and L_2 and ... and L_k
	Example	`[lastAddProperty(c1,name,'Vehicle'),` `lastAddProperty(c2,name,'Aircraft')]`
Disjunction	Syntax	`or`$[\mathtt{L}_1, \mathtt{L}_2, ..., \mathtt{L}_k]$ where $L_1, L_2, ..., L_k$ are *Atoms* or *Boolean Combinations*
	Semantics	L_1 or L_2 or ... or L_k
	Example	`or [lastAddProperty(c1,name,'Vehicle'),` `lastAddProperty(c1,name,'Aircraft')]`

Table 5.5 – Logic Operators - Boolean Combinations.

We also presented the new logic operators added to the representation language to increase the number of inconsistency types that Badger can describe.

6 Scalability

We assess the scalability of Badger in this chapter. For this purpose we use generated UML models, reverse engineering models and a toy example model. We gradually increase during our experiments the size of the model, the number of inconsistencies to resolve, and the number of generated plans. We conclude this chapter with a summary of the scalability analysis results.

The work presented in this chapter was previously published in: *"Badger: A regression planner to resolve design model inconsistencies."* in European Conference Modelling Foundations and Applications (ECMFA), 2012, Winner of the ECMFA 2012 Best Foundation Paper Award, co-authored with Tom Mens (University of Mons, Belgium) and Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium) [125].

6.1 Experimental Setup

Due to the unavailability of a sufficiently large sample of realistic UML models, we evaluate the scalability of Badger on generated UML models, as well as on UML models obtained by reverse engineering Java programs, and on a classical toy example. We consider for our experiments 13 structural model inconsistency types, listed in Table 6.1. They are based on the well-formedness constraints of the UML 2.3 metamodel expressed in OCL [10, 35, 117, 154, 164]). Each entry in Table 6.1 consists of an id followed by the metamodel element on which the constraint is specified in the UML Superstructure document [117]. Next, a short description of the inconsistency type is given, followed by the page number of the UML Superstructure document where the inconsistency type can be found.

All experiments reported in this chapter were carried on a Power Mac with 2.80GHz Intel Xeon Quad-Core processor and 4Gb of RAM. We used the 32-bit version of SWI-Prolog 6.0.2, running on the Ubuntu 11.04 operating system. All timing results obtained were averaged over 10 different runs to account for performance variations[1].

6.2 Generated Models

Due to the unavailability of a sufficiently large sample of realistic UML models, we use an existing model generator that was proposed, mathematically grounded and validated in Mougenot *et al.* [106]. The model generator implements a value generator for the model element's properties (*i.e.* names, literal values, visibility kinds, direction kinds) in order to produce valid models. It also implements two generators, for generalizations and for references, which randomly choose valid targets in the generated elements.

In order to generate inconsistent models, we modified the generators for properties, generalizations and references. For example, we customised generators to authorise lower bounder multiplicities smaller than 0 (I_5), to authorise lower bounder multiplicities bigger than the upper bounder multiplicities (I_6) and to authorise not-binary association being aggregations (I_1).

This model generator enables us to study the impact of the size of the models on the approach. It also enables us to apply our approach to a large set of models with a wide range of different sizes. The model generator creates class diagrams based on the UML metamodel shown in Figure 6.1.

We used the model generator to create 941 models with model sizes ranging from 21 to 10849 model elements (*i.e.*, elements obtained using the Praxis elementary operation `create`). Obviously, the generated models also contain references (from 21 to 11504) and properties (from 40 to 22903), obtained by using the elementary operations `addProperty` and `addReference`, respectively.

6.2.1 First Experiment

In a first experiment, we ran *Badger* on all generated models and computed the timing results for generating a single resolution plan. We analysed the relation between the number of model elements and the time (in milliseconds) needed to resolve only one inconsistency of a particular type. In order to compare the timing results for different inconsistency

[1] In hindsight, it would have been more appropied to use the minimal time instead of the average time since the noise generated by external factors will never be negative.

id	metamodel element	description of the inconsistency type (see [117])
I_1	Association	Only binary associations can be aggregations (p. 39)
I_2	Element	Elements that must be owned must have an owner (p. 65)
I_3	Named Element	If a NamedElement is not owned by a Namespace, it does not have a visibility (p. 101)
I_4	Multiplicity Element	A multiplicity must define at least one valid cardinality that is greater than zero. (p. 97)
I_5	Multiplicity Element	The lower bound must be a non-negative integer literal (p. 97)
I_6	Multiplicity Element	The upper bound must be greater than or equal to the lower bound (p. 97)
I_7	Classifier	The general classifiers are the classifiers referenced by the generalization relationships (p. 54)
I_8	Classifier	Generalization hierarchies must be directed and acyclic. A classifier can not be both a transitively general and transitively specific classifier of the same classifier (p. 54)
I_9	Classifier	A classifier may only specialize classifiers of a valid type (p. 54)
I_{10}	Property	A multiplicity on an aggregate end of a composite aggregation must not have an upper bound greater than 1 (p. 127)
I_{11}	Property	Only a navigable property can be marked as readOnly (p. 128)
I_{12}	Property	The value of isComposite is true only if aggregation is composite (p. 128)
I_{13}	Operation	An operation can have at most one return parameter (p. 107)

Table 6.1 – List of considered structural model inconsistency types.

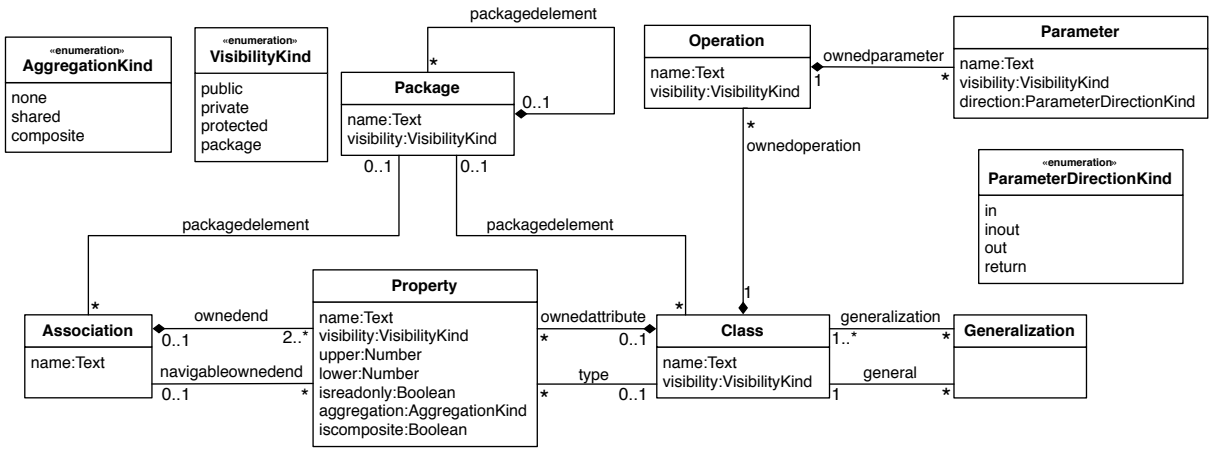


Figure 6.1 – Simplified fragment of the UML metamodel for class diagrams.

types, we repeated the experiment for each of the 13 considered inconsistency types shown in Table 6.1.

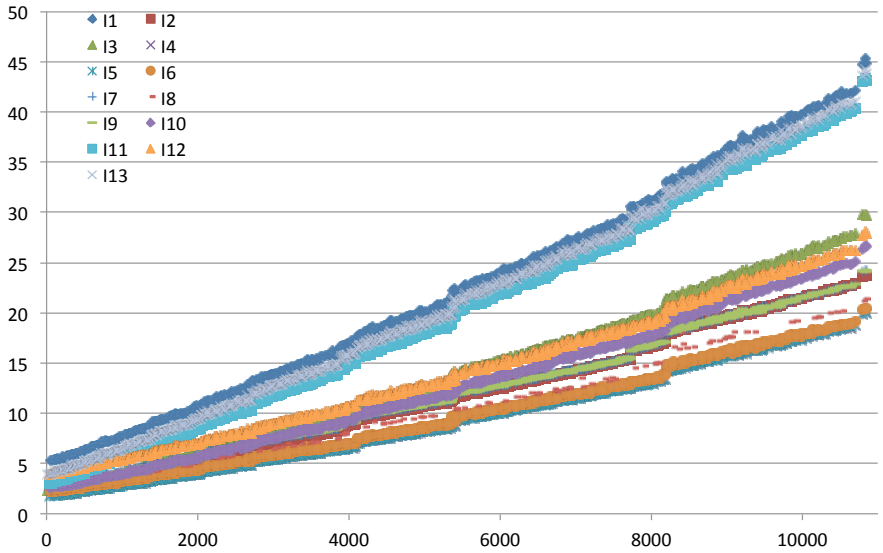


Figure 6.2 – Comparison of execution time (y-axis, expressed in milliseconds) per model size (x-axis, expressed as number of model elements) for resolving a single inconsistency in 941 different models. Different colours and symbols represent different inconsistency types.

The results of the experiment are visualised in Figure 6.2. The time needed to resolve inconsistencies of a particular inconsistency type mainly depends on the size of the model and on the number of logic literals in the desired goal. For example, I_{13} requires 4 literals and takes on average 4.2 times longer than I_5 that only uses 1 literal.

We fitted five different types of parametric regression models: a linear model, a logarithmic model, a quadratic polynomial model, a power model and an exponential model. The goodness of fit of each type of model was verified using the adjusted coefficient of determination $\bar{R}^2$. Its value is always between 0 and 1, and a value close to 1 corresponds to a good fit. Table 6.2 shows the obtained $\bar{R}^2$ values. In order to easily distinguish the best regression models, values higher than 0.90 are indicated in *italics*, while values higher than 0.95 are indicated in **boldface**. In addition, per inconsistency type the regression models with the highest $\bar{R}^2$ value are marked with (*).

By analysing Table 6.2 we observe that the logarithmic regression models provide the worst results. In contrast to the four other considered types of regression models, its $\bar{R}^2$ values are always lower than 0.8. For these reasons, we exclude this type of regression model from the remainder of the analysis of our results. Based on the $\bar{R}^2$ values, the linear and the polynomial quadratic models are the best in all cases (with an $\bar{R}^2 > 0.99$ for all quadratic models). The polynomial models have a growth very close to linear with parameters a close to 0. This can explain the excellent results for the linear model (with an $\bar{R}^2 > 0.98$ for all linear models). A visual interpretation also confirms that the linear models are a very good match. The exponential and power models are also good fits, with $\bar{R}^2$ values that are always close to or above 0.9.

6.2.2 Second Experiment

In a second experiment, we studied how the generation of resolution plans with *Badger* scales up when resolving multiple inconsistencies *of different types* together. For each

	Linear $y = a + b\ x$	Log $y = a + b\ \ln(x)$	Polynomial Quadratic $y = a\ x^2 + b\ x + c$	Power $y = a\ x^b$	Exponential $y = a\ e^{b\ x}$
I_1	**0.994**	0.763	**0.999** (*)	*0.921*	**0.964**
I_2	**0.992**	0.744	**0.997** (*)	*0.921*	**0.953**
I_3	**0.994**	0.721	**0.999** (*)	*0.911*	*0.948*
I_4	**0.996**	0.760	**0.998** (*)	*0.933*	**0.963**
I_5	**0.990**	0.736	**0.998** (*)	*0.917*	**0.961**
I_6	**0.989**	0.740	**0.998** (*)	*0.909*	**0.970**
I_7	**0.991**	0.740	**0.997** (*)	*0.906*	**0.964**
I_8	**0.984**	0.701	**0.993** (*)	0.874	**0.963**
I_9	**0.991**	0.740	**0.997** (*)	*0.906*	**0.964**
I_{10}	**0.992**	0.755	**0.999** (*)	*0.925*	**0.961**
I_{11}	**0.994**	0.745	**0.999** (*)	*0.933*	*0.942*
I_{12}	**0.992**	0.738	**0.999** (*)	0.890	**0.976**
I_{13}	**0.994**	0.740	**0.999** (*)	*0.915*	**0.953**

Table 6.2 – $\bar{R}^2$ values of five different parametric regression models used to fit the timing results of Figure 6.2.

considered model, we resolved together one inconsistency of each of the 13 inconsistency types. Because not all models have at least one inconsistency of inconsistency type I_8, during our analysis we distinguished between models containing 12 inconsistencies (excluding I_8) and models containing 13 inconsistencies.

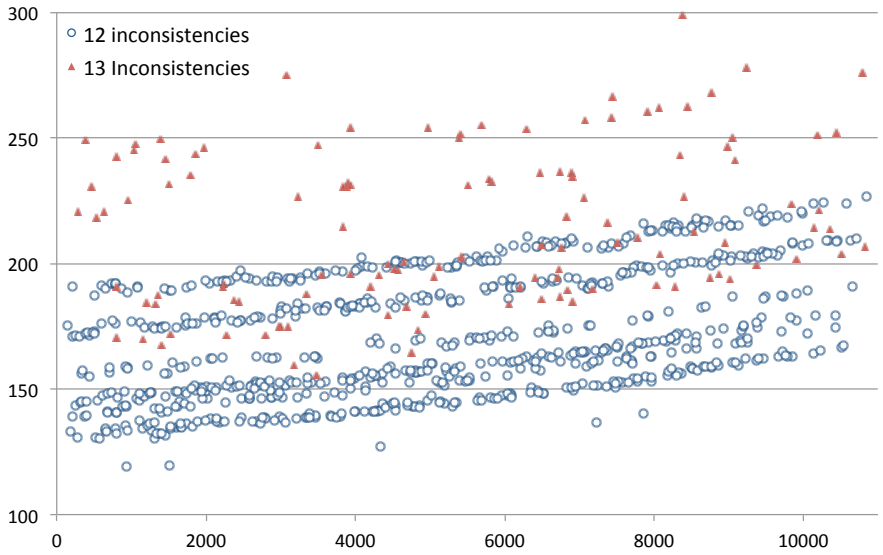


Figure 6.3 – Time comparison (y-axis, in milliseconds) per model size (x-axis, in number of model elements) for resolving multiple inconsistencies of different types in 941 different models.

Figure 6.3 presents the results of this experiment. The resolution time only increases slightly as the model size increases. None of the fitted regression models provide an $\bar{R}^2$ value higher than 0.25. As expected, the execution time is lower for 12 inconsistencies (mean = 170.91, median = 167.59) than for 13 inconsistencies (mean = 215.22, median = 211.48). Another factor that determines the execution time is the number of actions

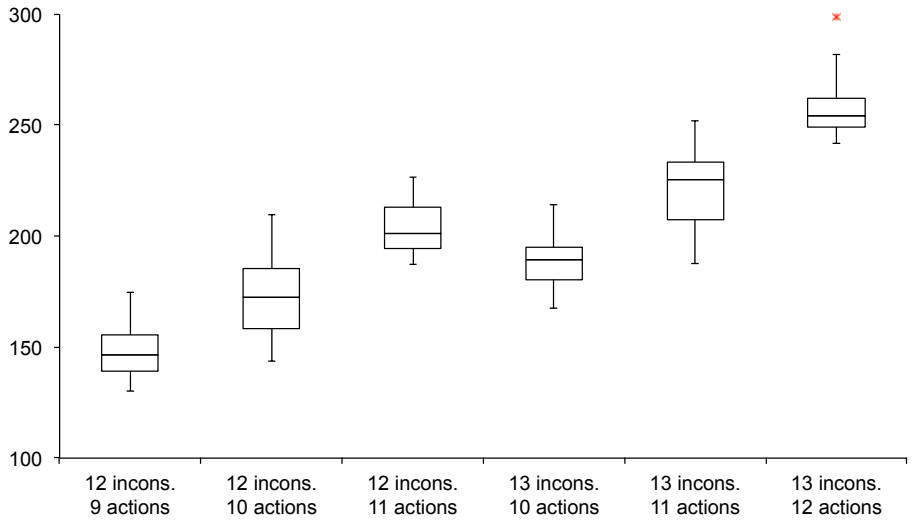


Figure 6.4 – Boxplots showing effect of number of actions on execution time (y-axis, in milliseconds).

in the resolution plan. Resolving 12 inconsistencies, requires between 8 and 11 actions (median = 10), while for 13 inconsistencies needs between 9 and 12 actions (median = 11). In addition, the resolution time increases as the number of actions increases, as shown in the box plots of Figure 6.4.

6.2.3 Third Experiment

In a third experiment, we studied how the generation of a resolution plan with *Badger* scales up if we want to resolve multiple inconsistencies *of the same type* together. To test this, we generated a very large model containing more than 10,000 elements and a large number of inconsistencies of each type. We excluded inconsistency type I_8 because the generated model does not contain enough inconsistencies of this type. For each of the remaining 12 inconsistency types we computed the time required to resolve an increasing number of inconsistencies (ranging from a single one to 70). Figure 6.5 visualizes the results.

As in Subsection 6.2.1, we fitted five types of parametric regression models: a linear model, a logarithmic model, a polynomial quadratic model, a power model and an exponential model. Table 6.3 shows the obtained $\bar{R}^2$ values. In order to easily distinguish the best regression models, values higher than 0.90 are indicated in *italics*, while values higher than 0.95 are indicated in **boldface**. In addition, per inconsistency type the regression models with the highest $\bar{R}^2$ value are marked with (*).

By analysing Table 6.3 we observe that the $\bar{R}^2$ was very high for 3 types of regression models (polynomial quadratic, power and exponential). This can visually be confirmed by the rapid increase of execution time as the number of inconsistencies increases. The quadratic models had the best fit, with an *adjusted* $R^2 > 0.98$ in all cases, followed by the exponential models (> 0.93 in all cases, and > 0.95 in 7 out of 12 cases).

The different growth rates observed in Figure 6.5 reflect the complexity of the inconsistency type to be resolved. For example, the inconsistency types whose resolution only requires the change of property values (*e.g.*, I_5 and I_6) take less time than those that need changes to references between model elements (*e.g.*, I_9 and I_7), because of the additional

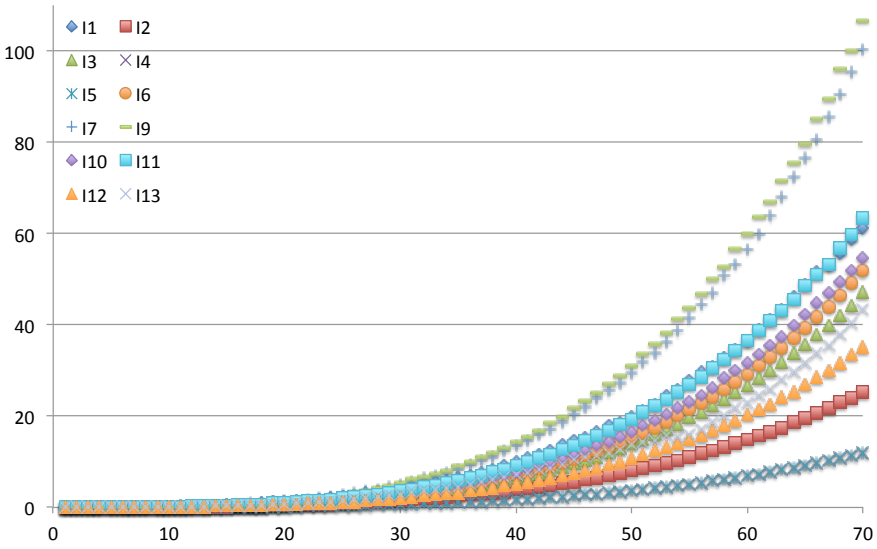


Figure 6.5 – Execution time (y-axis, in seconds) per number of inconsistencies of the same type (x-axis) for resolving multiple inconsistencies in a very large model. Different colours and symbols represent different inconsistency types.

	Linear $y = a + b\ x$	Log $y = a + b\ \ln(x)$	Polynomial Quadratic $y = a\ x^2 + b\ x + c$	Power $y = a\ x^b$	Exponential $y = a\ e^{b\ x}$
I_1	0.809	0.449	**0.991** (*)	*0.930*	*0.943*
I_2	0.799	0.439	**0.989** (*)	*0.916*	**0.957**
I_3	0.782	0.422	**0.986** (*)	*0.915*	**0.955**
I_4	0.789	0.429	**0.987** (*)	0.889	**0.972**
I_5	0.789	0.429	**0.987** (*)	0.889	**0.972**
I_6	0.779	0.419	**0.986** (*)	*0.926*	*0.947*
I_7	0.779	0.420	**0.985** (*)	*0.941*	*0.936*
I_9	0.780	0.420	**0.985** (*)	*0.942*	*0.934*
I_{10}	0.786	0.426	**0.987** (*)	*0.926*	*0.949*
I_{11}	0.792	0.432	**0.988** (*)	*0.922*	**0.953**
I_{12}	0.782	0.422	**0.986** (*)	*0.915*	**0.955**
I_{13}	0.801	0.445	**0.981** (*)	*0.907*	**0.952**

Table 6.3 – $\bar{R}^2$ values of five different parametric regression models used to fit the timing results of Figure 6.5.

multiplicity constraints required for the latter.

6.3 Reverse Engineered Models and a Toy Example

As previously mentioned, due to lack of access to sufficiently large and realistic UML models, we also obtained UML models by reverse engineering five Java open source software systems (Chipchat[2], UMLet[3], SweetRules[4], CleanSheets[5] and ThinWire[6]). These five models were previously used to evaluate a different model inconsistency resolution approach by Van Der Straeten *et al.* [156]. The reverse engineering was done through Together 2006 Release 2 for Eclipse[7] resulting in class diagrams and sequence diagrams.

In this section we will also use a classical toy example model. This model was created by Anne Keller [68] based on Briand *et al.* [14] and describes an automated teller machine (ATM). It was first introduced by Russell Bjork[8] and was previously used by different authors [7, 15, 135, 137]. The ATM model is composed by a class diagram, sequence diagrams and use case diagrams.

The metamodel used by these models is a subset of the UML metamodel composed of 3 diagram types (class diagram, sequence diagram and use case diagram); 44 meta-classes; 156 meta-class properties; and 564 meta-model relationships. This metamodel is significantly bigger than the one used in the previous section (Figure 6.1).

Table 6.4 shows the model sizes expressed in the number of model elements and in the number of classes for the 6 models that we will use for the experiments in this section. Each model m_i was consistent, *i.e.*, none of the 13 inconsistencies were detected. To carry out our experiments, we manually added one inconsistency of each type to each of the 6 models.

id	Kind	Name	Version	Model Elements	Classes
m_1	Toy Example	ATM		1002	18
m_2	Reverse Engineered	Chipchat	1.0 beta 2	2113	14
m_3	Reverse Engineered	UMLet	9beta	4897	184
m_4	Reverse Engineered	SweetRules	2.0	6345	186
m_5	Reverse Engineered	CleanSheets	1.4b	7392	254
m_6	Reverse Engineered	ThinWire	1.2	8571	231

Table 6.4 – Reverse engineered models and a toy example

6.3.1 First Experiment

In a first experiment, we have run *Badger* on all six models and computed the timing results for generating a single resolution plan. We analysed the relation between the number of model elements and the time (in milliseconds) needed to resolve only one inconsistency

[2] `http://chipchat.sourceforge.net/`

[3] `http://www.umlet.com/`

[4] `http://sweetrules.projects.semwebcentral.org/`

[5] `http://sourceforge.net/projects/csheets/`

[6] `http://thinwire.sourceforge.net/`

[7] Together 2006 is an integrated modeling environment with support for reverse engineering of source code, see `http://techpubs.borland.com/together/tec2006/en/readme.html`

[8] `http://www.math-cs.gordon.edu/courses/cs211/ATMExample/`

of a particular type. In order to compare the timing results for different inconsistency types, we repeated the experiment for each of the 13 considered inconsistency types of Table 6.1.

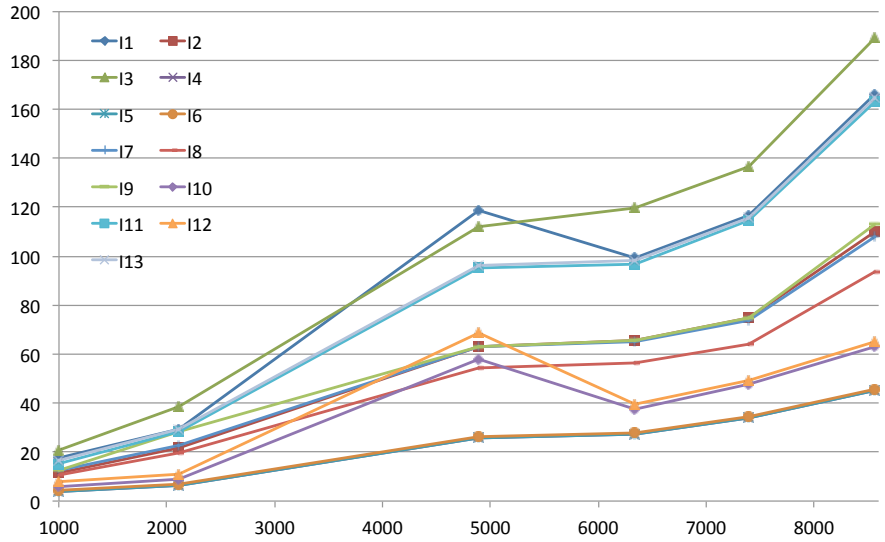


Figure 6.6 – Comparison of execution time (y-axis, expressed in milliseconds) per model size (x-axis, expressed as number of model elements) for resolving a single inconsistency in the 6 models. Different colours and symbols represent different inconsistency types.

The results of the experiment are visualised in Figure 6.6. Due to a lack of data points (only 6), it is impossible to determine a trend, neither visually or by fitting the resolution time to parametric regression models. Nevertheless, we calculated, for each model, the time for resolving one inconsistency type relative to the other inconsistencies types of that model. We observed that the relative resolution time is different per inconsistency type and it is quite similar across models (see Figure 6.7). As previously said, the time needed to resolve inconsistencies of a particular inconsistency type mainly depends on the size of the model and on the number of logic literals in the desired goal.

6.3.2 Second Experiment

In a second experiment, we studied how the generation of *multiple resolution plans* with *Badger* scales up when resolving multiple inconsistencies of different types together. To test this, for each considered model, we resolved together one inconsistency of each of the 13 inconsistency types, and we generated gradually from a single resolution plan to 100 resolution plans. Figure 6.8 presents the results of this experiment.

We fitted five types of parametric regression models: linear, logarithmic, polynomial quadratic, power and exponential. Table 6.5 shows the obtained $\bar{R}^2$ values. In order to easily distinguish the best regression models, values higher than 0.90 are indicated in *italics*, while values higher than 0.95 are indicated in **boldface**. In addition, per inconsistency type the regression models with the highest $\bar{R}^2$ value are marked with (*).

By analysing Table 6.5 we observe that the $\bar{R}^2$ was very high for the 3 types of models (polynomial quadratic models, exponential models and linear models). This can visually be confirmed by the rapid increase of execution time as the number of generated plans increases. The quadratic regression models had the best fit, with an *adjusted* $R^2 > 0.999$ in all cases, followed by the exponential regression models (> 0.95 in all cases).

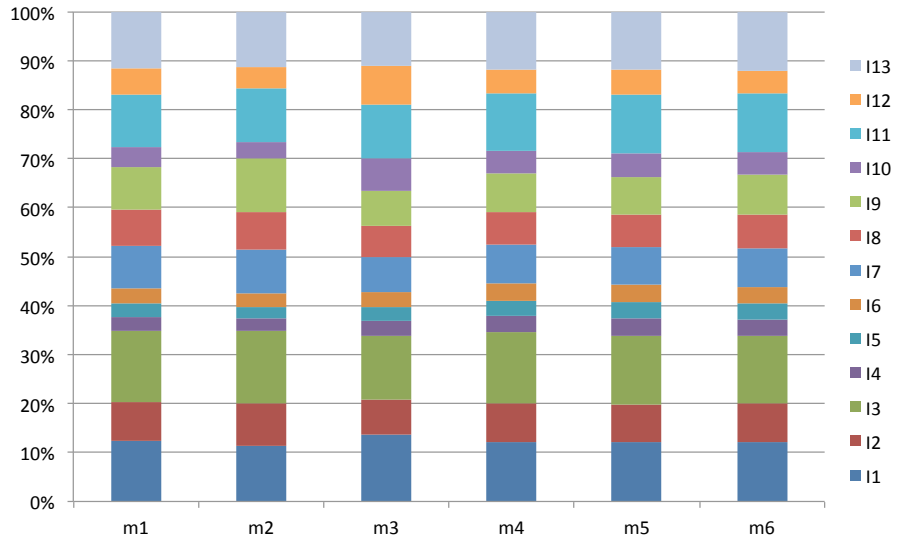


Figure 6.7 – Comparison of the relative execution time (y-axis, expressed in percentage) per model (x-axis, the model id) for resolving a single inconsistency in the 6 models. Different colours and symbols represent different inconsistency types.

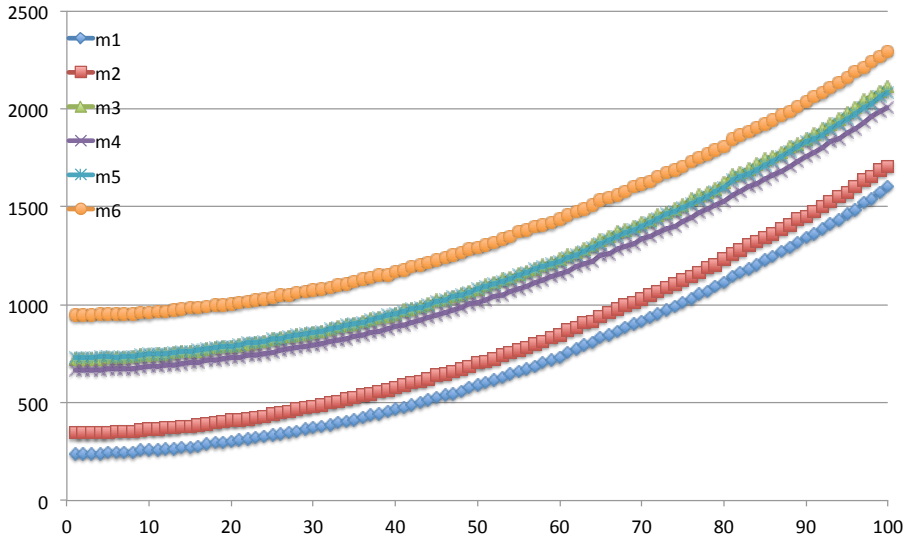


Figure 6.8 – Execution time (y-axis, expressed in milliseconds) per number of generated plans (x-axis) for 13 inconsistencies (one inconsistency per inconsistency type) in the 6 models. Different colours and symbols represent different models.

	Linear $y = a + b\ x$	Log $y = a + b\ \ln(x)$	Polynomial Quadratic $y = a\ x^2 + b\ x + c$	Power $y = a\ x^b$	Exponential $y = a\ e^{b\ x}$
m_1	*0.942*	0.606	**1.000** (*) $y = 0.1306x^2 + 0.5238x + 237.56$	0.769	**0.997**
m_2	*0.943*	0.608	**1.000** (*) $y = 0.1303x^2 + 0.6485x + 343.65$	0.745	**0.995**
m_3	*0.942*	0.607	**1.000** (*) $y = 0.1336x^2 + 0.5433x + 733.82$	0.700	**0.985**
m_4	*0.941*	0.605	**1.000** (*) $y = 0.1301x^2 + 0.4202x + 664.68$	0.701	**0.986**
m_5	*0.941*	0.605	**1.000** (*) $y = 0.131x^2 + 0.427x + 728.76$	0.695	**0.984**
m_6	*0.941*	0.605	**1.000** (*) $y = 0.1298x^2 + 0.482x + 944.51$	0.681	**0.979**

Table 6.5 – $\bar{R}^2$ values of five different parametric regression models used to fit the timing results of Figure 6.8.

The chart in Figure 6.8 may mislead us to think that the 6 models have an almost identical growth, while this is not the case. This is due to the similar value of the parameters a of the quadratic polynomial models (approximatively 0.13). But the parameters b are distinctively different. Remember that two quadratic polynomials have the same growth if they are parallel, and to be parallel they need to have the same parameters a and b.

6.4 Summary of the scalability analysis

We have evaluated the scalability of Badger by performing 3 experiments on 941 automatically generated UML class diagram models of various sizes and by running as well 2 experiments on UML models obtained by reverse engineering five Java programs and on a classical toy example. We used a set of 13 structural inconsistency types based on OCL constraints found in the UML metamodel specification. To stress-test Badger, we have increased one parameter at a time: the size of the model, the number of inconsistencies to resolve and the number of generated plans.

Our approach for resolving inconsistencies scales up to models containing more than 10000 model elements. The execution time appears to be linear or quadratic when the size of the model increases, polynomial quadratic when the number of inconsistencies to resolve increases, and polynomial quadratic when the number of generated plans increases. The execution time also increases as the number of actions in the resolution plan increases.

The aim of this chapter was not only to stress-test Badger, but also to demonstrate that Badger is capable of resolving different inconsistency types in different models of varying sizes.

7

Evaluation Function Analysis

As explained in Chapter 4, Badger uses a recursive best-first search (RBFS) to explore the state space by expanding the most promising node. RBFS evaluates the most promising node by using an *evaluation function*, which is defined as the sum of an heuristic function and a cost function: $f(n) = h(n) + g(n)$.

In this chapter we analyse and discuss the solutions and possibilities we have explored to allow Badger to adapt the way it presents the inconsistency resolution plans to the users, by changing the order in which the resolution plans are generated. Firstly we present and discuss the heuristic function. We evaluate and illustrate by means of examples how changes in the cost function impact the order in which the resolution plans are generated. We conclude this chapter by a discussion about the possibilities of Badger to present the resolution plans to the user.

In this chapter, we analyse and discuss the solutions and possibilities we have explored to adapt the way in which Badger presents the resolution plans to the users, by changing the order in which the resolution plans are generated. The grey part of Figure 7.1 highlights the parts of the algorithm that we are going to study in this chapter: *the evaluation function*. In Section 7.1 we will briefly present and discuss *the heuristic function*. In Section 7.2 we will evaluate how *the cost function* impacts the order in which the resolution plans are generated.

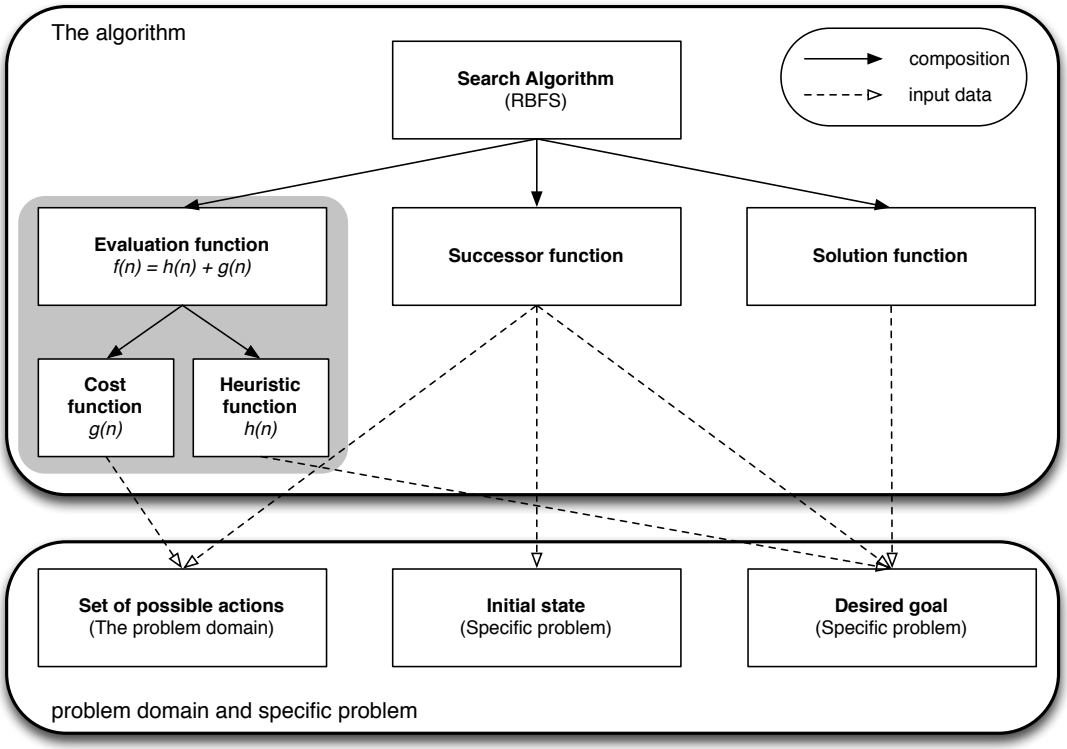


Figure 7.1 – A block diagram representing the functions which compose Badger. The grey part represents the evaluation function that will be changed in this chapter.

7.1 The heuristic function

The *heuristic function* $h(n)$ is the minimal estimation of the cost to reach a solution from the node n. It can be derived by defining a *relaxed problem* that is easier to solve.

Among the different readily-available heuristics, the one used by Badger is a well-known planner heuristic that *ignores the preconditions*. It fulfils the objectives of this thesis, because it produces good results in inconsistency resolution as shown in Chapter 6. Although it is possible to change and explore other planner heuristics in Badger, developing such new heuristics takes a lot of effort and trial and error and is therefore out of the scope of this dissertation. Nevertheless, in case someone would like to explore a new heuristic, we can provide some recommendations. As we have explained in Chapter 3, the heuristic functions implemented in classical planning are typically non-admissible. Those heuristics that are admissible are poorly informed and the application of optimal search algorithms is too expensive in terms of computation time. Therefore, we recommend to use an heuristic based on the common relaxations in automated planning (*e.g.*, ignore the delete effect, subgoal independence) [65].

7.2 The cost function

The cost function $g(n)$ used by Badger defines the user-specified cost of applying each action. This cost affects the order in which the plans are generated.

In all experiments that we have conducted in previous chapters, Badger used the same cost for each action, implying that the generated plans are ordered in terms of the number of actions they contain: the shortest plans are generated first. When multiple plans have the same cost, the order in which they are generated depends on the order in which the set of possible actions and the model elements are indexed by Prolog.

We will show in Section 7.2.1 that, contrary to what one might expect, these shortest plans are not necessarily those plans that delete model elements. We will also counter another common misconception that one of the possible plans is a plan that deletes all model elements.

In Section 7.2.2 we will illustrate by means of examples how the user can change the cost function to affect the order in which the resolution plans are generated. This allows users to adapt the way in which Badger presents the resolution plans.

7.2.1 Two common questions about the plans generated by Badger

In this section we will try to answer two common questions about the plans generated by Badger: **(i)** is the plan with *delete* actions always the shortest plan ?; and **(ii)** is *deleting all model elements* a possible plan ?

Let us begin with question **(i)**. Plans that delete model elements do not necessarily contain less actions because Badger takes into account the metamodel to avoid ill-formed models. And taking into account these metamodel constraints implies using additional actions. For example, a relationship can only exist between two existing model elements. That means that, before deleting a model element, Badger needs to take care of all the relationships involved with this model element. In fact, only deleting a reference with zeros as lower multiplicities in the metamodel or deleting an optional property takes only one action. In all other cases, the constraints of the metamodel force Badger to involve additional actions.

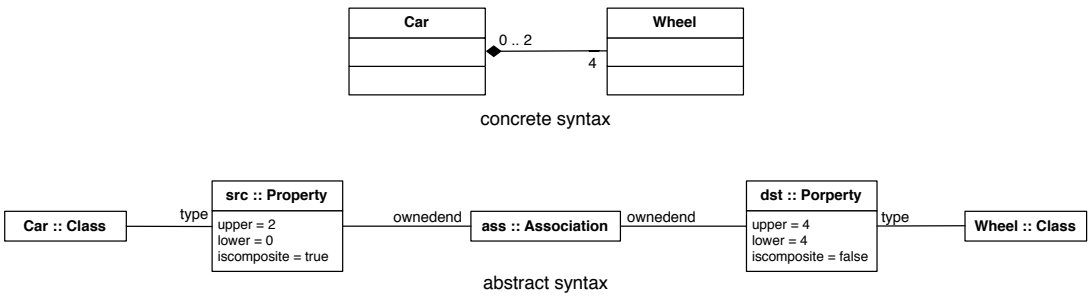


Figure 7.2 – Concrete and abstract syntax of a small class diagram with an inconsistency of type I_{10} (*cf.* Table 6.1).

To illustrate the above, Figure 7.2 shows the concrete and abstract syntax of a small model with two classes and an association between them. The model contains an inconsistency of type *"A multiplicity on an aggregate end of a composite aggregation must not have an upper bound greater than 1."* (I_{10} of Table 6.1). This inconsistency arises because

the upper bound of the aggregated end (0..2) in the composite association between `Car` and `Wheel` is greater than 1.

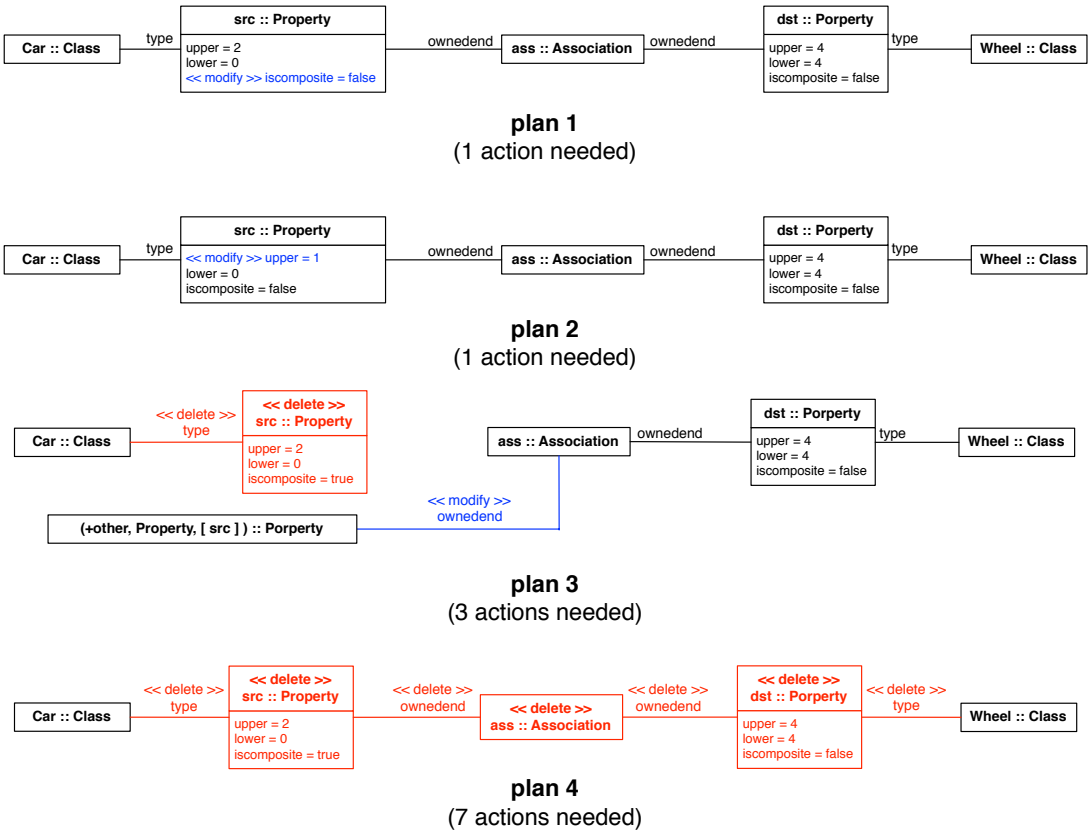


Figure 7.3 – The 4 resolution plans proposed by Badger to solve the inconsistency found in Figure 7.2.

Figure 7.3 graphically shows the 4 resolution plans proposed by Badger. Note that plan 4 deleting the association is the last one that is generated as it contains more actions than the other ones (7 actions). To delete the association's end `src`, we must first deal with its two references (`type` and `ownedend`). The `type` reference can simply be deleted while the `ownedend` reference can be modified to point to a different association end (plan 3) or deleted. If the `ownedend` reference is deleted, the association must also be deleted because the metamodel specifies that an association has minimum two association ends. Deleting the association implies deleting the other association end, because when a composite is deleted, all of its parts must be deleted with it (remember that an `Association` is composed by its `Association Ends`). After 7 actions, the association is finally correctly removed to result in a well-formed model.

Let us now address the second question : is *deleting all model elements* a possible plan ? This question comes from the assumption that we want to remove some specific inconsistencies and empty plans do not have any inconsistencies, so an empty plan seems to be a solution.

The answer is no, in most cases. The reasoning is as follows. Badger generates plans using only relevant actions and stops when the inconsistencies are solved. A *relevant action* is an action that contributes to the resolution of the inconsistency. Badger uses a *solution function* to check if all given inconsistencies are solved (*cf.* Chapter 4). Badger only performs the actions needed to remove the given inconsistencies. This means that the only way for Badger to delete all the model elements is when all the model elements

are part of one or more inconsistencies and removing all the model elements is mandatory to solve the inconsistencies. For example, an inheritance cycle in a class diagram can be solved by removing only one generalization. Therefore, Badger will never propose to delete all the generalizations to take care of the inconsistency since the removal of one generalization suffices.

7.2.2 Changing the cost function

In this section, we illustrate the effect of the order in which Badger generates plans when the cost function is changed to give higher priority to certain resolution plans.

Changing the order in which the resolution plans are presented to the user is not a novel idea, Küster and Ryndina [79] introduce the concept of side-effect expressions to determine whether or not a resolution introduces a new inconsistency. They attach a cost to each inconsistency type to compare alternative resolutions for the same inconsistencies.

The cost function of Badger takes as parameter the kind of action as well as the elementary model operation that is going to be changed by the action (*cf.* Section 5.2). The following information can be used to define or modify the cost function and change the order in which Badger generates the resolution plans: **(i)** the kind of action; **(ii)** the model element on which the elementary model operation is performed; **(iii)** the element of the metamodel; **(iv)** the author that performed the elementary model operation; and **(v)** the revision in which it has been performed.

Hitherto, Badger used the same cost for each action, which lead to the generation of shortest plans first. If the user wants to adapt the order in which Badger generates the plans, she can change the cost function to give priority to one kind of plan or to another. To do this, she can use the previously listed information to define the cost function. For instance, she can assign different costs to the different *kinds of actions* to give different priorities to them. In this section, we illustrate several examples in which we change the cost function to give priorities **(a)** to certain kinds of actions; **(b)** to certain parts of the metamodel; **(c)** to certain model elements; **(d)** to certain authors; and **(e)** to a combination of the kind of action and the kind of metamodel.

a) Kind of action priorities

In this example, we illustrate the effect of the order in which Badger generates plans when the cost function is changed to give higher priority to certain kind of actions. To illustrate this, we use a simple class with an inconsistency of type: *"an operation can have at most one return parameter"* (I_{13} of Table 6.1). Figure 7.4 shows the concrete and the abstract syntax of the class. The inconsistency arises when the operation `getDiameter()` has two return parameters (one float and one integer).

Badger proposes the following 10 plans, generated and presented by increasing order of actions, to resolve this inconsistency.

1. modify the reference `ownedparameter` between the operation `getDiameter` and the parameter `p1` to make the parameter belong to a different operation;
2. modify the reference `ownedparameter` between the operation `getDiameter` and the parameter `p2` to make the parameter belong to a different operation;
3. modify the property `direction` of the parameter `p1` from `return` to `in`;
4. modify the property `direction` of the parameter `p1` from `return` to `inout`;
5. modify the property `direction` of the parameter `p1` from `return` to `out`;

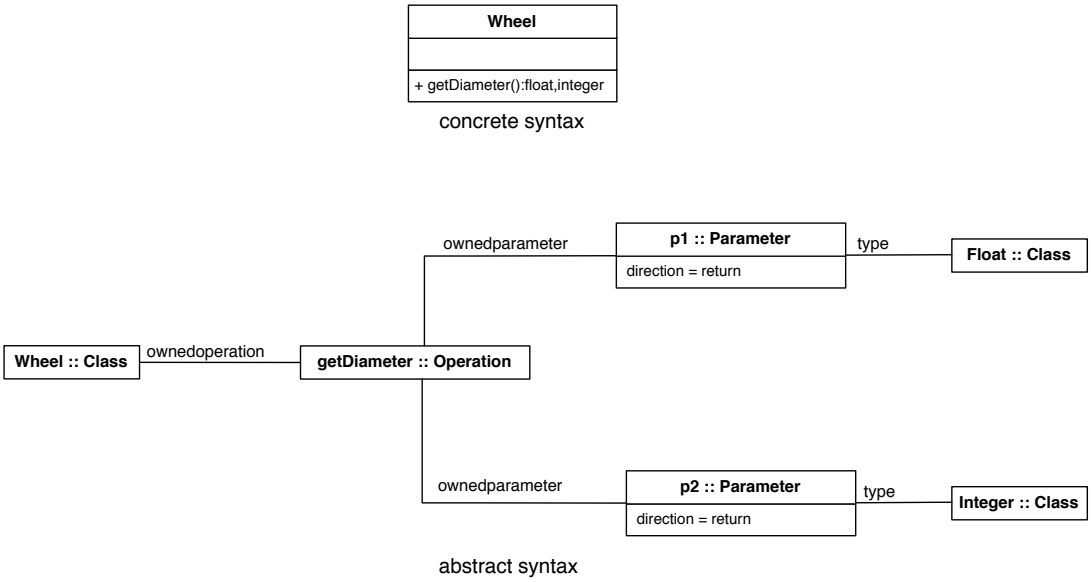


Figure 7.4 – Concrete and abstract syntax of a class with an inconsistency of type I_{13} (*cf.* Table 6.1).

6. modify the property `direction` of the parameter `p2` from `return` to `in`;
7. modify the property `direction` of the parameter `p2` from `return` to `inout`;
8. modify the property `direction` of the parameter `p2` from `return` to `out`;
9. delete the parameter `p1` and delete also the references: `ownedparameter` between the parameter `p1` and the operation `getDiameter`, and the reference `type` between the parameter `p1` and the class `Float`;
10. delete the parameter `p2` and delete also the references: `ownedparameter` between the parameter `p2` and the operation `getDiameter`, and the reference `type` between the parameter `p2` and the class `Integer`.

To assign a higher priority to certain kind of actions, the user needs to change the cost function and Badger will generate the resolution plans with this new cost function. We illustrate this by means of three scenarios: (1) the user gives higher priority to the plans with delete actions; (2) the user gives higher priority to the plans with delete actions and lower priority to the plans with actions that modify the references; and (3) the user gives higher priority to the plans with delete actions and she does not want the plans with actions that modify the references. Table 7.1 shows the order in which the plans are generated depending on the chosen cost function.

In scenario (1), the user changes the cost of the delete actions (delete model element, delete reference and delete property) to be smaller than the rest of the action costs (1 for the delete actions and 5 for the rest). This gives a higher priority to the plans with delete actions, so these plans will be generated first. Using this new cost function, Badger generates the same plans but in a different order, the two first plans are now plans 9 and 10 (see column 2 of Table 7.1). Each one of these two plans deletes one of the parameters: `p1` and `p2`, respectively.

In scenario (2), the user changes the cost of the actions that modify the references to be higher than the rest of the action costs, and the cost of delete actions to be the lowest (10 for the actions that modify the references, 1 for the delete actions and 5 for the rest). This moves the plans using actions that modify references to the last positions while the plans with delete actions will come first. Using these new costs, Badger generates the

Default settings: Shortest plans first	Scenario (1): Highest priority to plans with delete actions	Scenario (2): Highest priority to plans with delete actions and lowest priority to plans with actions that modify references	Scenario (3): Highest priority to plans with delete actions and never generate plans with actions that modify references
1	9	9	9
2	10	10	10
3	1	3	3
4	2	4	4
5	3	5	5
6	4	6	6
7	5	7	7
8	6	8	8
9	7	1	
10	8	2	

Table 7.1 – Order in which the resolution plans for resolving the inconsistencies in Figure 7.4, are generated depending on the chosen cost function with kind of actions priority.

plans in a new order in which plans 1 and 2 are the last generated plans and plans 9 and 10 are the first.

In scenario (3), the user changes the cost of the actions that modify the references to infinite (∞ for the actions that modify the references, 1 for the delete actions and 5 for the rest). An infinite cost in Prolog is represented by the cost function returning `fail`. This prohibits the generation of plans that use these actions. Using this new cost function, Badger generates only 8 plans, leaving out plans 1 and 2 that involve modifying references.

b) Metamodel Priority

In this example, we illustrate the effect of the order in which Badger generates plans when the cost function is changed to give higher priority to certain parts of the metamodel. To illustrate this we use a simple model composed of a class diagram and a sequence diagram. This model has an inconsistency of type: *"each message in a sequence diagram needs to have a corresponding operation that needs to be owned by the message receiver's class"* (rule R_1 in Chapter 1). Figure 7.5 shows the concrete syntax of this model and the abstract syntax of the inconsistent part of the model. The inconsistency arises when the message `play` used in the sequence diagram is not defined by an operation in the class `Streamer`.

Badger proposes the following 8 plans, generated and presented by increasing order of actions, to resolve this inconsistency.

1. modify the reference `sentTo` between the message `play` and the lifeline `st: Streamer` so that the message is sent to a different lifeline;
2. modify the property `name` of the message `play` from `play` to `stream`;
3. modify the property `name` of the operation `stream` from `stream` to `play`;
4. modify the property `name` of the message `play` from `play` to `wait`;
5. modify the property `name` of the operation `wait` from `wait` to `play`;

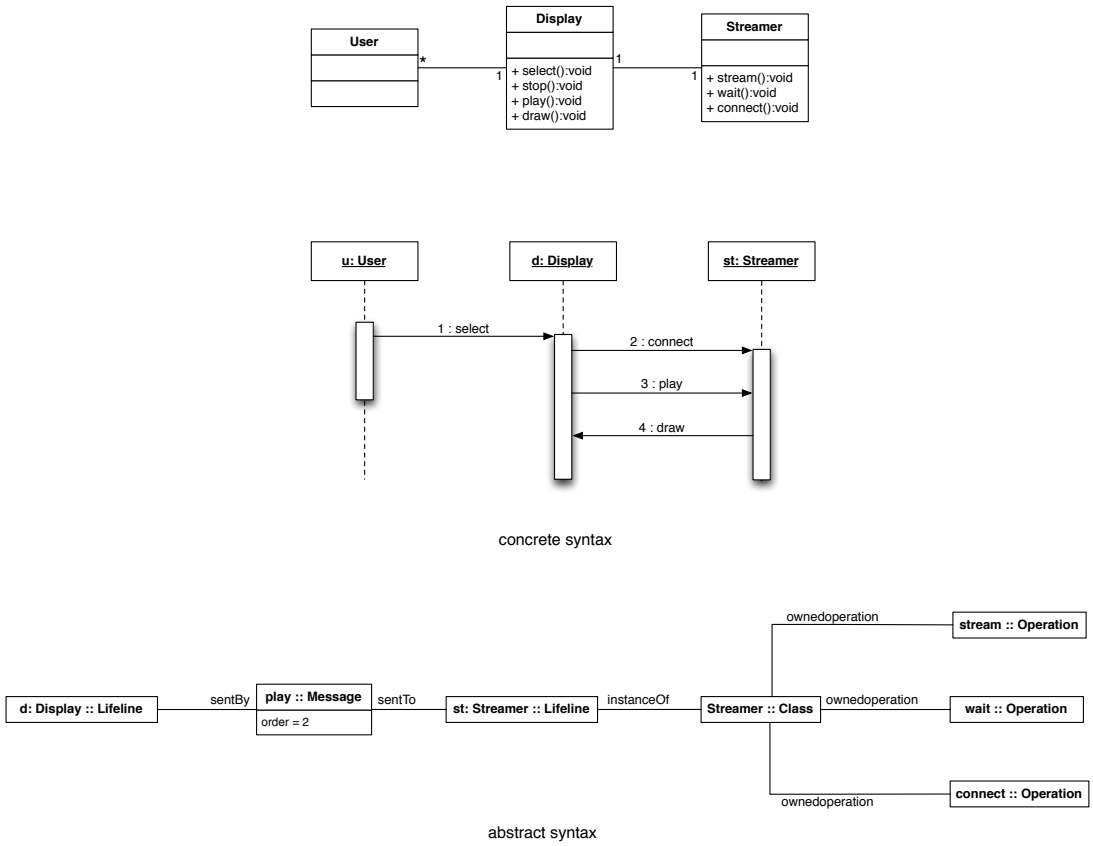


Figure 7.5 – Concrete and abstract syntax of a small class diagram and sequence diagram with an inconsistency of type *"each message in a sequence diagram needs to have a corresponding operation that needs to be owned by the message receiver's class"* (rule R_1 in Chapter 1).

Default settings: Shortest plans first	Scenario (1): Plans with actions that modify the sequence diagram first	Scenario (2): Plans with actions that modify the class diagram first	Scenario (3) Never generates plans with actions that modify the class diagram
1	1	3	1
2	2	5	2
3	4	7	4
4	6	1	6
5	8	2	8
6	3	4	
7	5	6	
8	7	8	

Table 7.2 – Order in which the resolution plans for resolving the inconsistencies in Figure 7.5, are generated depending on the chosen cost function with metamodel priority.

6. modify the property `name` of the message `play` from `play` to `connect`;
7. modify the property `name` of the operation `connect` from `connect` to `play`;
8. delete the message `play` and also delete the references: `sentBy` between the lifeline `d: Display` and the message `play` and the reference `sentTo` between the message `play` and the lifeline `st:Streamer`.

To assign a higher priority to certain parts of the metamodel, the user needs to change the cost function and Badger will generate the resolution plans with this new cost function. We illustrate this by means of three scenarios: (1) the user gives higher priority to the sequence diagram; (2) the user gives higher priority to the class diagram; and (3) the user does not want the plans that manipulate the class diagram. Table 7.2 shows the order in which the plans are generated depending on the chosen cost function.

In scenario (1), the user changes the cost functions to give higher priority to the sequence diagram. This is done by assigning a smaller cost (value 1) to the actions that create, modify or delete an object belonging to the sequence diagram compared to the cost of the rest of the actions (value 5). Using this new cost function, Badger generates the same plans in a different order: the five plans generated first involve the manipulation of an object belonging to the sequence diagram, while the three plans generated last involve the manipulation of an object belonging to the class diagram (see column 2 of Table 7.2).

In scenario (2), the user changes the cost function to give higher priority to the class diagram by assigning a smaller cost (value 1) to the actions that create, modify or delete an object belonging to the class diagram compared to the rest of the action costs (value 5). The new order of generated plans is shown in column 3 of Table 7.2: the first three plans that manipulate an object belonging to the class diagram.

In scenario (3), the user changes the cost function to avoid generating plans that modify the class diagram. This is done by assigning an infinite cost (∞) to the actions that create, modify or delete an object belonging to the class diagram. Using this new cost function, Badger generates only 5 plans (see column 4 of Table 7.2) leaving out the plans that modify the class diagram (plans 3, 5 and 7).

c) Model Priority

In this example, we will illustrate the effect of the order in which Badger generates plans when the cost function is changed to give higher priority to certain model elements. For

Default settings: Shortest plans first	Scenario (1): Plans that do not modify or delete the message `play` and its relationships first	Scenario (2): Plans that modify the operations `wait` and `connect` first	Scenario (3): Never generate plans with actions that modify or delete the message `play` and its relationships
1	3	5	3
2	5	7	5
3	7	1	7
4	1	2	
5	2	3	
6	4	4	
7	6	6	
8	8	8	

Table 7.3 – Using model priorities to change the order in which the resolution plans for resolving the inconsistencies in Figure 7.5 are depending on the chosen cost function.

this we will use same model as the one that was used in the previous example (Figure 7.5).

To assign a higher priority to certain model elements, the user needs to change the cost function and Badger will generate the resolution plans with this new cost function. We illustrate this by means of three scenarios: (1) the user gives less priority to the plans that modify or delete the message `play`; (2) the user gives higher priority to the plans that modify or delete the operations `wait` and `connect`; and (3) the user does not want the plans that modify or delete the message `play`. Table 7.3 shows the order in which the plans are generated depending on the chosen cost function.

In scenario (1), the user changes the cost function to give a bigger cost to the actions that modify or delete the message `play` or its relationships: `sentBy` and `sentTo` (5 for these actions and 1 for the rest of actions). This is done to generate firstly all kinds of plans that do not modify the message `play` (see column 2 of Table 7.3).

In scenario (2), the user changes the cost function to give more importance to the plans that modify the operations `wait` and `connect`. This is done by assigning a small value (1) to the actions that modify or delete the operations `wait` and `connect` than to the other actions (5). The new order of plans starts by the plan that modify the name `wait` into `play` and by the plan that modify the name of `connect` into `play` (see column 3 of Table 7.3).

In scenario (3), the user changes the cost function to never modify the `play` message or its relationships: `sentBy` and `sentTo`. This is done by assigning an infinite cost (∞) to the actions that modify or delete the message or its relationships. Using this new cost function, Badger generates only 3 plans, see column 4 of Table 7.3, leaving out the plans that modify the message or its relationships (plans 1, 2, 4, 6 and 8).

d) Author Priorities

In this example, we illustrate the effect of the order in which Badger generates plans when the cost function is changed to give priority to certain authors.

As example we will use a simple class diagram with an inconsistency of type *"Generalization hierarchies must be directed and acyclic. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier"* (I_8 of Table 6.1). Fig-

ure 7.6 shows the concrete and abstract syntax of this model. The inconsistency arises because of an inheritance cycle between the classes `Vehicle`, `Aircraft` and `Airplane`. Note that three authors participated in the creation of this class diagram: `Tom`, `Jorge` and `Mathieu`. The modifications performed by `Tom` are visually expressed in black in Figure 7.6, the modifications performed by `Jorge` in orange and the modifications performed by `Mathieu` in purple. Note also that the inconsistency was not introduced by one individual person but by the combination of the work of `Tom` and `Jorge`. So, it corresponds in fact to a structural syntactic merge conflict [92].

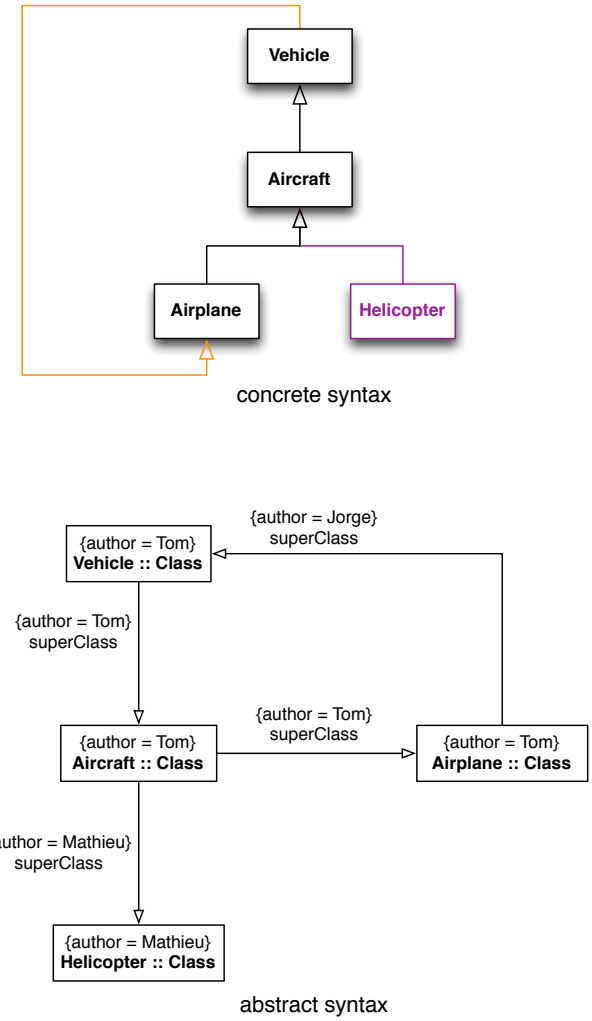


Figure 7.6 – Concrete and abstract syntax of a small class diagram with an inconsistency of type I_8 (*cf.* Table 6.1).

Badger proposes the following 3 plans, generated and presented by increasing order of actions, to resolve this inconsistency:

1. delete the reference `superClass` (created by Tom) between the classes `Vehicle` and `Aircraft`;
2. delete the reference `superClass` (created by Jorge) between the classes `Airplane` and `Vehicle`;
3. delete the reference `superClass` (created by Tom) between the classes `Aircraft` and `Airplane`;

To assign a higher priority to certain authors, the user needs to change the cost function and Badger will generate the resolution plans with this new cost function. To illustrate this we create four scenarios: (1) the user gives higher priority to the plans that modify

Default settings: Shortest plans first	Scenario (1): Plans with actions that modify the part of Jorge first	Scenario (2): Only plans with actions that modify the part of Jorge	Scenario (3): Plans with actions that modify the part of Mathieu first	Scenario (4): Only plans with actions that modify the part of Mathieu
1	2	2	1	
2	1		2	
3	3		3	

Table 7.4 – Changing author priorities to change the order in which the resolution plans for resolving the inconsistencies in Figure 7.6 are generated, depending on the chosen cost function.

the part of the model created by `Jorge`; (2) the user only wants plans that modify the part of the model created by `Jorge`; (3) the user gives higher priority to the plans that modify the part of the model created by `Mathieu`; and (4) the user only wants plans that modify the part of the model created by `Mathieu`. Table 7.4 shows the order in which the plans are generated depending on the chosen cost function.

In scenario (1), the user changes the cost function to give higher priority to the plans with actions that modify the part of the model created by `Jorge`. This is done by assigning a smaller cost (value 1) to the actions that modify or delete a model element or reference introduced by `Jorge` compared to the rest of action costs (value 5). Using this new cost function, Badger generates the same plans but in a different order: the first generated plan (plan 2) is the only one that involves a manipulation of a reference introduced by `Jorge` (see column 2 of Table 7.4).

In scenario (2), the user changes the cost function to only generate plans that involve a manipulation of model elements or references created by `Jorge`. This is done by assigning an infinite cost (∞) to the actions that create, modify or delete a model element or reference that has not been created by Jorge. Using this new cost function, Badger generates only a single plan (plan 2) as it is the only plan that involves a manipulation of a reference introduced by `Jorge` (see column 3 of Table 7.4).

In scenario (3), the user changes the cost function to give a higher priority to the plans with actions that modify the part of the model created by `Mathieu`. This is done by assigning a smaller cost to the actions that modify or delete a model element or reference introduced by `Mathieu`. Using this new cost function, Badger generates the same plans in the same order as with the default settings (see column 4 of Table 7.4). This is because none of the plans has actions that modify or delete a model element belonging to `Mathieu`, given the fact that the model elements belonging to `Mathieu` are not part of the inconsistency.

In scenario (4), the user changes the cost function to only generate plans that involve a manipulation of model elements or references created by `Mathieu`. This is done by assigning an infinite cost (∞) to the actions that create, modify or delete a model element or reference that has not been created by `Mathieu`. Using this new cost function, Badger does not generate any plan (see column 5 of Table 7.4). This is because no plan has actions that modify or delete a model element belonging to `Mathieu`. With this example we observe that, when an infinite cost is used, Badger may not return any plan.

e) Combining Priorities

In this example, we will illustrate how we can combine different kind of priorities to change the effect of the order in which Badger generates plans. The cost function is changed to give higher priority to certain kind of actions as well as to certain kind of metamodel element.

As example we will use a simple class diagram with an inconsistency of type: *"Inherited Cyclic Composition" (ICC)* [151]. Figure 7.7 shows the concrete and abstract syntax of this model. An ICC inconsistency occurs when a composition relationship and an inheritance chain form a cycle that would produce an infinite containment of objects upon instantiation. The inconsistency appears in the inheritance chain `Vehicle` ← `Car` ← `Amphibious Vehicle` and in the composition relationship between `Vehicle` and `Amphibious Vehicle`.

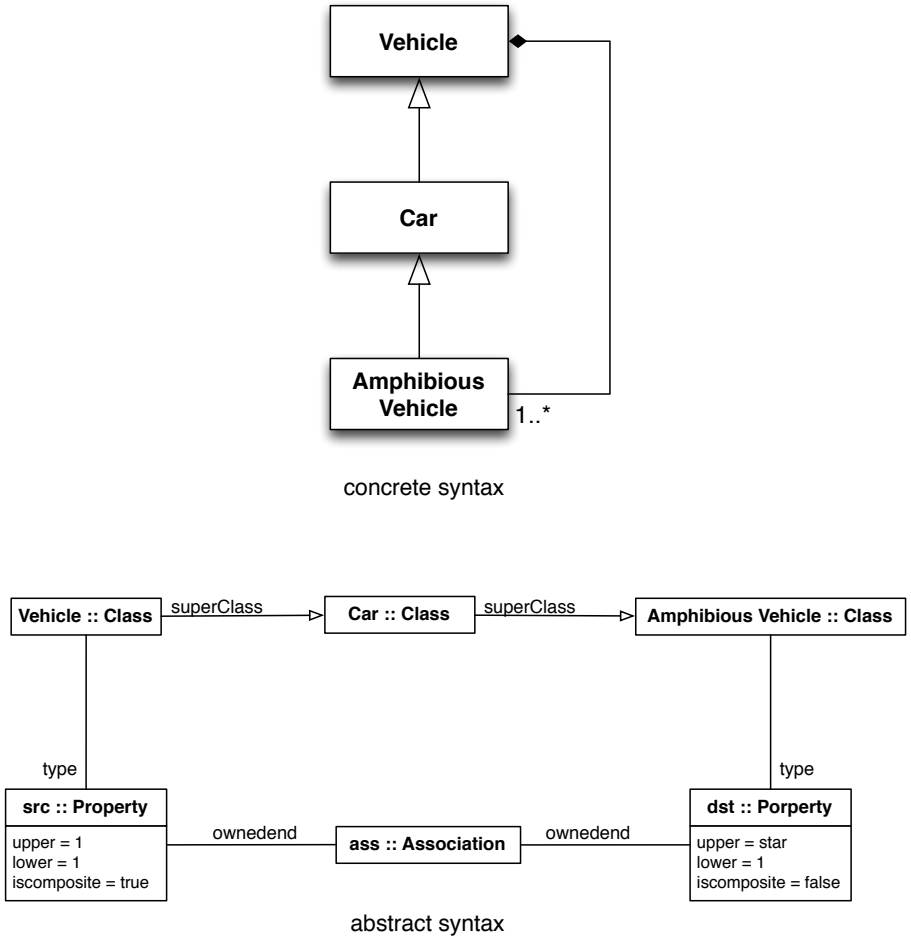


Figure 7.7 – Concrete and abstract syntax of a small class diagram with an inconsistency of type *"Inherited Cyclic Composition" (ICC)* [151].

Badger proposes the following 9 plans, generated and presented by increasing order of actions, to resolve this inconsistency.

1. delete the reference `superClass` between the classes `Vehicle` and *Car*;
2. delete the reference `superClass` between the classes `Car` and `Amphibious Vehicle`;
3. modify the reference `type` between the class `Vehicle` and the property (association end) `src` so the last one point to a different class;

4. modify the reference `type` between the class `Amphibious Vehicle` and the property (association end) `dst` so the last one point to a different class;
5. modify the `iscompostie` property of the property (association end) `src` from `true` to `false`;
6. modify the `lower` property of the property (association end) `dst` from 1 to 0;
7. delete the property `src`, also delete the reference between the property `src` and the class `Vehicle`, and modify the reference between the property `src` and the association `ass` so the latter can own a different property.
8. delete the property `dst`, also delete the reference between the property `dst` and the class `Amphibious Vehicle`, and modify the reference between the property `dst` and the association `ass` so the latter can own a different property.
9. delete the association `ass`, ass well as both properties `src` and `dst` and the references `type` and `ownedend`.

To assign a higher priority to certain kind of actions as well as to certain kind of metamodel element, the user needs to change the cost function and Badger will generate the resolution plans with this new cost function. To illustrate this we create three scenarios: (1) the user gives less priority to the plans that manipulate the `superClass` kind of reference; (2) the user gives less priority to the plans that manipulate the `superClass` kind of reference and gives higher priority to all kind of actions except the one that modify the references; and (3) the user gives priority to all kind of actions except the one that modify the references and does not want to generate plans that manipulate the `superClass` kind of references. Table 7.5 shows the order in which the plans are generated depending on the chosen cost function.

In scenario (1), the user changes the cost function to assign a bigger cost to the actions that manipulate `superClass` references (cost 10 for these actions and 1 for the other actions). This is done to generate firstly all kinds of plans that do not modify or delete any `superClass` reference. Using this new cost function Badger generates the same plans but with a different order. The plans that delete the reference of kind `superClass` (plans 1 and 2) are generated last (see column 2 of Table 7.5).

In scenario (2), the user changes the cost function to give a medium cost to the actions that modify references (cost 5 for the actions that modify references, 10 for the actions that manipulate `superClass` references and 1 for the rest of actions). This generates firstly all kind of plans that do not modify references and that do not modify or delete any `superClass` reference. Next the plans that modify references are generated and finally the plans that modify or delete the `superClass` reference are generated.

In scenario (3), the user changes the cost function to give an infinite cost to the actions that manipulate `superClass` references (∞ for the actions that manipulate `superClass` references, 5 for the actions that modify references, and 1 for the rest of actions). Badger will generates only 7 plans, see column 4 of Table 7.5, leaving out the plans that delete the `superClass` references (plans 1 and 2).

7.3 Discussion

In this chapter we presented and discussed about the heuristic function. We also presented the cost function and answered two common questions about the plans generated by Badger. We illustrated by the mean of examples, how changing the cost function affects the generated plans. There are different information that can be used by the user to

Default Settings: Shortest plans first	Scenario (1): Less priority to plans that manipulate the `superClass` kind of reference	Scenario (2): Less priority to plans that manipulate the `superClass` kind of reference and priority to all kind of actions except ones that modify references	Scenario (3): Priority to all kind of actions except the ones that modify references and never generate plans that manipulate the `superClass` kind of reference
1	3	5	5
2	4	6	6
3	5	9	9
4	6	3	3
5	7	4	4
6	8	7	7
7	9	8	8
8	1	1	
9	2	2	

Table 7.5 – Order in which the resolution plans are generated for resolving the inconsistencies in Figure 7.7, depending on the chosen cost function with kind of actions and metamodel priority.

change the cost function: action kind, metamodel, model element, author and revision. The user can also combine these as it was illustrated in our last example.

In order to make the approach useful in practice, the resolution that the user actually prefers should be one of the first generated resolution plans. We presented in this chapter how to change the cost function to allow Badger to adapt the way it presents the resolution plans to the users. We also illustrated how some kind of resolution plans can be omitted by attaching an infinite weight to the cost function. However, controlled user studies are still needed to assess what would be the most suitable cost function in practice, for each particular need. This is left as future work.

8

Conclusions and Future Work

This chapter concludes the dissertation by summarising the contributions and obtained results. It also discusses the limitations and presents the open research perspectives.

8.1 Contributions

In this dissertation we used automated planning, a logic-based approach originating from artificial intelligence, for the purpose of automatically generating model inconsistency resolutions. We are not aware of any other work having used this technique for this purpose.

We started by a feature-based analysis of design model inconsistency resolution approaches based on three main criteria: flexibility, usability and extensibility. This study allowed us to identify weaknesses in eight recent approaches and to take into account these weaknesses to avoid them in the development of our own solution.

We presented two different planning approaches to generate resolution plans: *Fast-Forward Planning System* (FF), an existing domain-independent heuristic state-space progression planner; and *Badger*, a new domain-specific regression planner that we have implemented in Prolog. We studied their feasibility in the domain of model inconsistency resolution. While FF does the job, it suffers from scalability and has very poor timing results. *Badger* has demonstrated good performance for inconsistency resolution, is metamodel-independent and can generate multiple resolutions plans. In addition, the planner algorithm of Badger can be adapted by taking full advantage of the domain knowledge.

Badger requires as input a model and a set of inconsistencies. In contrast to other inconsistency resolution approaches, the planner does not require the user to specify resolution rules manually or to specify information about the causes of the inconsistency. To specify models in a metamodel-independent way, and to be able to reuse an existing model generator, we relied on the Praxis language [10].

We validated Badger on 941 automatically generated UML class diagram models of various sizes (from 21 to 10849 model elements), as well as on UML models obtained by reverse engineering five Java programs (from 2113 to 8571 model elements), and on a classical toy example (1002 model elements). We used a set of 13 structural inconsistency types based on OCL constraints found in the UML metamodel specification. We analysed the scalability results of the approach obtained through several stress-tests and discussed the limitations of our approach. Our empirical analysis revealed a strong linear relationship between the model size and the execution time, a quadratic relationship between the number of inconsistencies and the execution time and a quadratic relationship between the number of generated plans and the execution time. We also observed an increase of the execution time as the number of actions in the resolution plans increases. In addition, our approach scales up to models containing more than 10000 model elements. The metamodel independence was validated by applying Badger to the problem of resolving code smells in Java programs.

We conclude that it is feasible to use automated planning in a scalable way for design-model inconsistency resolution. Nevertheless, improvements could still be made to this approach and in addition, this dissertation opens up new research directions that can be explored in more detail in the future.

8.2 Threats to Validity and Limitations

Our approach has only been stress-tested on a Java program and on UML models consisting of class diagrams, sequence diagrams and use case diagrams. However, the fact that

we rely on a metamodel independent representation (using sequences of Praxis elementary model operations) makes it straightforward to apply it to other types of structural models as well. We restricted ourselves to represent only structural aspects of models with Praxis. We considered only a limited set of inconsistency types, but we have taken into account a variety of logic operators and elementary model operations. It remains an open question whether and how the approach can be generalised to non-structural inconsistencies (*e.g.*, concrete syntax inconsistencies, behavioural inconsistencies).

Only the inconsistencies that can be expressed using the logic operators presented in Tables 5.4 and 5.5 can be used in our approach. If one wants to express inconsistencies using other operators, the approach needs to be changed in order to take into account these new operators. Because the new operators can have an impact on the timing results and on the performance of the approach, an equilibrium should be found between the performance and the expressiveness.

Despite all the validations and testing we carried out, Badger may still contain some bugs we are not aware of.

To carry out our experiments, we relied on an external model generator [106]. This may cause a bias as the generated models may not look like "real" models. This bias is limited since the model generator we used relies on the Boltzmann random sampling method that generates, in a scalable way, uniform samplings of any given size. We also carry out our experiments using reengineered models and a classical toy example, but it remains an open question whether the approach can scale up into industrial ("real") models.

In order to make the approach useful in practice, the resolution that the user prefers most should be one of the first generated resolution plans. The order in which resolution plans are generated can be modified easily by modifying the cost function of the planner algorithm, as shown in Chapter 7. In addition, entire resolution plans can be omitted by attaching an infinite weight to certain actions. Assessing what would be the most suitable parameters for the cost function in practice requires a controlled user study, which is left as future work. In order to make the approach usable in practice, it should be integrated into an integrated modeling environment (a CASE tool). It also should integrate an intuitive visual interface allowing the user to easily adapt the cost function to her needs without requiring any programming experience or knowledge of the underlying planning algorithm.

Our approach does not take into account derived metamodel elements. For example, the `isComposite` property can be derived from the `AggregationKind` property: if the *AggregationKind* has as value `composite` the *isComposite* property takes as value `true`, otherwise it takes as value *false*. Inconsistency definitions can refer to a derived metamodel element or to an original metamodel element. When the resolution needs to change a derived element, not only this element must be changed but also the element it is derived from. When the resolution needs to change an original element, its derived element must also be changed. To take into account the derived elements, the metamodel and the set of actions of our approach need to be changed to perform not only changes to the target element but also to its original element and to its derived elements.

Our approach works with the assumption that the inconsistency rules are correct, *i.e.*, that no false positive inconsistencies are detected in the model. To resolve inconsistencies in presence of false positives, the approach must be able to identify semi-automatically if the detected inconsistency was produced because there is a real inconsistency in the model (true positive) or whether it was produced because there is a problem in the detection

rule (false positive) (*cf.* Castro et. al. [20]).

8.3 Future Work

Our first priority in the near future is to address the current limitations of our approach. In particular, controlled user studies are needed to study the quality of the resolution plans as perceived by the user. An empirical classification of the inconsistency types by their resolution time is also needed. This classification could be helpful to predict the necessary time to resolve the detected inconsistencies. An empirical classification of the inconsistency types by the logic operators used to specify the inconsistency types could also be interesting.

It could be interesting to use automated planning in model refactoring, another model-driven software engineering challenge. A model refactoring is a model transformation that needs to preserve the behaviour of the model and model transformations. The main challenge here is to find how to describe and preserve the behaviour in models. Automated planning has already been used by Javier Pérez [121] in order to generate refactoring strategies in source code (Java programs). This approach could be used in model refactoring, but the planner domain knowledge, refactoring rules and refactoring strategies, need to be added manually. The other way around, Badger could be adapted to be used in model refactoring, but a way to describe and preserve behavioural of models must be added.

In this dissertation, we used classical planning, an automated planning variant, for the purpose of automatically generating model inconsistency resolution. In a future work we would like to study if it is feasible to use *planning under uncertainty* [28, 160], a different variant of automated planning. *Planning under uncertainty* aims to resolve problems in a nondeterministic and *partially observable* world, in which the planner has incomplete knowledge of the current state. It remains an open question if *planning under uncertainty* can be used to resolve inconsistencies in distributed models. Distributed models are used in the context of collaborative work in industrial systems involving hundreds of developers working on hundreds of models. In distributed models, the models are continuously edited by developers who work asynchronously on their local copy, and commit from time to time their work to the rest of the developers [105]. The planner, resolving inconsistencies locally, must also take into account the possibility to share information (*e.g.*, resolution plans, model elements) with the other planners installed somewhere else.

Even if automated planning met our expectations, we would like to study other techniques coming from the domain of artificial intelligence for the purpose of resolving modeling inconsistencies in an automated way.

Logic-based approaches have been used for different but related purposes in inconsistency resolution. Marcelloni and Akist [85,86] used *fuzzy logic* to cope with methodological inconsistencies in design models. It remains to be seen whether this approach can be generalised to resolve any kind of model inconsistency. Castro *et al.* [20] used *logic abduction* to detect and resolve inconsistencies in source code. We have started to carry out some promising experiments to apply this approach to resolve inconsistencies in design models appeared promising, but a full-fledged experimental study is necessary to assess whether the approach actually scales up and works in practice.

Harman [57] advocates the use of search-based approaches in software engineering. This includes a wide variety of different techniques and approaches such as metaheuristics (*e.g.*, variable neighborhood search [18, 19]), local search algorithms, automated learning,

genetic algorithms [129]. We believe that these techniques could be applied to the problem of model inconsistency resolution, because it satisfies at least three important properties that motivate the need for search-based software engineering: the presence of a large search space, the need for algorithms with a low computational complexity, and the absence of known optimal solutions. How these search-based approaches would compare to our proposed automated planning approach and to existing model inconsistency resolution approaches, remains an open question.

In order to assess the adequacy of all these different approaches to inconsistency resolution, there is also an urgent need to define benchmarks allowing to compare them. Such a benchmark should contain at least a set of shared case studies on which to evaluate each approach; as well as a set of clearly identified criteria enabling the comparison of approaches and their quality. These benchmarks would allow us to carry out an experimental study of different approaches. Defining such benchmarks is a challenge of its own because the existing model inconsistency resolution approaches that can be found in research literature are quite diverse. They are implemented in different programming languages and on different operating systems, and use different input formats, modeling languages and metamodels for the modes, different input and output formats for inconsistencies and resolutions. The benchmark should also impose a fixed set of inconsistency rules, which is quite difficult if the modeling languages or the metamodels are different, that need to be resolved by each resolution approach. Analysing the resolution quality of the different approaches would require a controlled user study. Indeed, in presence of a wide variety of different resolution approaches, the only meaningful notion of quality would be the quality as perceived by the user.

Bibliography

[1] Marcus Alanen and Ivan Porres. A metamodeling language supporting subset and union properties. *Software and System Modeling*, 7(1):103–124, 2008.

[2] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards automated inconsistency handling in design models. In *Proceedings of the 22st International Conference on Advanced Information Systems, CAISE'10*, volume 6051 of *Lecture Notes in Computer Science*, pages 348–362. Springer, June 2010.

[3] F. Baader, D. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[4] Robert Balzer. Tolerating inconsistency. In *Proceedings of the International Conference Software Engineering (ICSE)*, volume 1, pages 158–165. IEEE Computer Society, 1991.

[5] Robert Balzer. "Tolerating Inconsistency" revisited. In *Proceedings of the International Conference Software Engineering (ICSE)*, page 665. IEEE Computer Society, May 2001.

[6] Kent Beck and Martin Fowler. *Bad Smells in Code*, chapter 3. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1st edition, June 1999.

[7] Umesh Bellur and V. Vallieswaran. On OO design consistency in iterative development. In *Proceedings of the 3rd International Conference on Information Technology: New Generations*, pages 46–51. IEEE Computer Society, 2006.

[8] D. Bernard, E Gamble, N. Rouquette, B. Smith, Y. Tung, N. Muscettola, G. Dorias, B. Kanefsky, J. Kurien, W. Millar, P. Nayak, and K. Rajan. Remote Agent Experiment DS1 Technology Validation Report. *NASA Ames Research Center and JPL*, 1998.

[9] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. A canonical scheme for model composition. In *Proceedings on Model Driven Architecture - Foundations and Applications, Second European Conference (ECMDA-FA)*, pages 346–360, Bilbao, Spain, July 10-13 2006.

[10] Xavier Blanc, Alix Mougenot, Isabelle Mounier, and Tom Mens. Detecting model inconsistency through operation-based model construction. In Robby, editor, *Proceedings of the International Conference Software engineering (ICSE'08)*, volume 1, pages 511–520, Leipzig, Germany, May 10-18 2008. ACM.

[11] Xavier Blanc, Alix Mougenot, Isabelle Mounier, and Tom Mens. Incremental detection of model inconsistencies based on model operations. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 2009.

[12] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642, 1995.

[13] Ivan Bratko. *Prolog programming for artificial intelligence*. Addison-Wesley, 2001.

[14] L.C. Briand, Y. Labiche, and L. O'Sullivan. Impact Analysis and Change Management of UML Models. Technical Report SCE-03-01, Carleton University, 2003.

[15] Lionel C. Briand, Yvan Labiche, L. O'Sullivan, and Michal M. Sówka. Automated impact analysis of UML models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[16] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 20th anniversary edition, 1995.

[17] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, March 1998.

[18] Jason Brownlee. Variable neighbourhood search. Technical Report CA-TR-20100206-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, February 2010.

[19] Gilles Caporossi and Pierre Hansen. Variable Neighborhood Search for Extremal Graphs: 1. The AutoGraphiX System. *Discrete Mathematics*, 212(1-2):29–44, 2000.

[20] Sergio Castro, Johan Brichau, and Kim Mens. Diagnosis and semi-automatic correction of detected design inconsistencies in source code. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST)*, pages 8–17, New York, NY, USA, 2009. ACM.

[21] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *European Conference on Planning*, 1997.

[22] Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. In *FASE*, volume 2029 of *Lecture Notes in Computer Science*, pages 17–31. Springer, April 2001.

[23] Aurélie Clodic, Maxime Ransan, Rachid Alami, and Vincent Montreuil. A management of mutual belief for human-robot interaction. In *Proceedings of the International Conference Systems, Man and Cybernetics*. IEEE, 2007.

[24] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of programming languages—II*, pages 331–367, New York, NY, USA, 1996. ACM.

[25] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Feature Modeling*, chapter 5, pages 83–116. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[26] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[27] Gregory De Fombelle. *Gestion incrémentale des propriétés de cohérence structurelle dans l'ingénierie dirigée par les modèles*. PhD thesis, Université Pierre et Marie Curie, Paris, 2007.

[28] T. Dean, K. Basye, R. Chekaluk, S. Hyun, M. Lejter, and M. Randazza. Coping with uncertainty in a control system for navigation and exploration. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI)*, volume 2, pages 1010–1015, Boston, 1990. MIT Press.

[29] Stuart E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17:395–412, 1969.

[30] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating distributed ViewPoints: the Anatomy of a Consistency Check. *Concurrent Engineering*, 2(3):209, 1994.

[31] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2004.

[32] Alaxender Egyed. Instant consistency checking for UML. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 381–390. ACM Press, 2006.

[33] Alexander Egyed. Fixing inconsistencies in UML design models. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 292–301. IEEE Computer Society, 2007.

[34] Alexander Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.

[35] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Proceedings ACM/IEEE International Conference Automated Software Engineering (ASE '08)*, pages 99–108, New York, NY, USA, 2008. ACM.

[36] M. Elaasar and L. Brian. An overview of UML consistency management. *Technical Report SCE-04-18*, August 2004.

[37] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Testing the consistency of dynamic UML diagrams. In *Proceedings of the International Conference Integrated Design and Process Technology (IDPT)*, June 2002. Pasadena, CA, USA.

[38] G. Engels, R. Heckel, J.M. Küster, and L. Groenewegen. Consistency-Preserving Model Evolution through Transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proceedings of the International Conference Unified Modeling Language (UML)*, pages 212–227. Springer, 2002.

[39] Jean-Marie Favre. Languages evolve too! changing the software time scale. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 33–44, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[40] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *L'ingénierie dirigée par les modèles*. Hermes - Lavoisier, 2006.

[41] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Company, 1998.

[42] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[43] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the International Joint Conference Artificial Intelligence*, pages 608–620, 1971.

[44] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. In *IEEE Transactions on Software Engineering*, volume 20, pages 569–578. IEEE Computer Society Press, 1994.

[45] A. Finkelstein, G. Spanoudakis, and D. Till. Managing interference. In *Foundations of Software Engineering*, pages 172–174. ACM, 1996.

[46] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[47] P. Fradet, D. Le Metayer, and M. Peiin. Consistency checking for multiple view software architectures. In *Proceedings Joint Conference ESEC/FSE'99*, volume 41, pages 410–428. Springer, September 1999.

[48] Héctor Geffner. Functional Strips: a more flexible language for planning and problem solving. *Logic-based Artificial Intelligence*, pages 188–209, 2000.

[49] Alfonso Gerevini and Derek Long. BNF description of PDDL 3.0. http://zeus.ing.unibs.it/ipc-5/, October 2005.

[50] Alfonso Gerevini and Derek Long. Plan Constraints and Preferences in PDDL3. Technical Report R. T. 2005-08-47, Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia, 2005.

[51] M. Ghallab, D.S. Nau, and P. Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann Publishers, 2004.

[52] Michael Goedicke, Torsten Meyer, and Gabriele Taentzer. Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In *Proceedings of the International Symposium Requirements Engineering*, pages 92–99. IEEE Computer Society, 1999.

[53] Bas Graaf and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *Proceedings of the International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES)*, pages 115–126. IEEE Computer Society, 2007.

[54] John C. Grundy, John G. Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.

[55] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. *Handbook of Graph Grammars and Computing by Graph Transformation*, 2:607–638, 1999.

[56] David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Weizmann Science Press of Israel, Jerusalem, Israel, 2000.

[57] Mark Harman. Search based software engineering. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume 3994/2006 of *Lecture Notes in Computer Science*, pages 740–747. Springer Berlin / Heidelberg, 2006. Workshop on Computational Science in Software Engineering (CSSE'06).

[58] P. E. Hart, N. J. Nilsson, and B. Raphale. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[59] J.H. Hausmann, R. Heckel, and S. Sauer. Extended Model Relations with Graphical Consistency Conditions. In L. Kuzniarz, G. Reggio, J.L. Sourrouille, and Z. Huzar, editors, *Blekinge Institute of Technology, Research Report 2002:06. UML 2002 Workshop on Consistency Problems in UML-Based Software Development*, pages 61–74, 2002.

[60] A. Hessellund, K. Czarnecki, and A. Wasowski. Guided development with multiple domain-specific languages. *Lecture Notes in Computer Science*, 4735:46, 2007.

[61] Jörg Hoffmann. FF: The Fast-Forward Planning System. *The AI Magazine*, 2001.

[62] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[63] Ellis Horowitz and Sartaj Sahni. *Fundamentals of computer algorithms*. Computer Science Press, 1978.

[64] Michaël Hoste, Jorge Pinna Puissant, and Tom Mens. Challenges in model-driven software evolution in the 7th belgian-netherlands software evolution workshop (benevol) 2008 workshop. Technical Report CS-Report 08-30, Technische Universiteit Eindhoven, December 2008.

[65] Sergio Jiménez Celorrio. *Planning and Learning under Uncertainty*. PhD thesis, Universidad Carlos III de Madrid, 2010.

[66] M.D. Johnston. Spike: AI scheduling for NASA's hubble space telescope. In *Sixth Conference on Artificial Intelligence Applications*, pages 184–190, 1990.

[67] H. A. Kautz and B. Selman. Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[68] Anne Keller. *Analysis-based Resolution Support for Inconsistencies in UML Models.* PhD thesis, Universiteit Antwerpen, 2012.

[69] J.-P. Kelly, A. Botea, and S. Koenig. Offline planning with hierarchical task networks in video games. In *Proceedings of the International Conference Artificial Intelligence and Interactive Digital Entertainment*, pages 60–65, 2008.

[70] Mathias Kleiner, Marcos Didonet Del Fabro, and Patrick Albert. Model search: Formalizing and automating constraint solving in MDE platforms. In *Proceedings Modelling Foundations and Applications, 6th European Conference, ECMFA*, pages 173–188, Paris, France, June 2010.

[71] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an adl subset. In Sam Steel and Rachid Alami, editors, *Recent Advances in AI Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 273–285. Springer Berlin / Heidelberg, 1997.

[72] R. E. Korf. Depth-first iterative-deepening an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[73] R. E. Korf. Iterative-deepening A*: An optimal admissible tree search. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1034–1036, 1985.

[74] R. E. Korf. Best-first search with limited memory. *UCLA Computer Science Annual*, 1991.

[75] R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[76] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[77] Thomas Kühne. Matters of (meta-) modeling. *Software and System Modeling*, 5:369–385, 2006.

[78] J.M. Kuster and G. Engels. Consistency Management Within Model-Based Object-Oriented Development of Components. *Lecture Notes in Computer Science*, 3188:157–176, 2004.

[79] J.M. Kuster and K. Ryndina. Improving Inconsistency Resolution with Side-Effect Evaluation and Costs. *Lecture Notes in Computer Science*, 4735:136, 2007.

[80] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 2006.

[81] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology (EWSPT)*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[82] S Lin. Computer solutions of the travelling salesman problem. *Bell Systems Technical Journal*, 44(10):2245–2269, 1965.

[83] WenQian Liu, Steve Easterbrook, and John Mylopoulos. Rule-based detection of inconsistency in UML models. In *Proceedings of the UML Workshop on Consistency Problems in UML-based Software Development*, pages 106–123. Blekinge Insitute of Technology, 2002.

[84] H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *Proceedings Symposium on Applied computing (SAC '06)*, pages 1804–1809, New York, NY, USA, 2006. ACM.

[85] Francesco Marcelloni and Mehmet Aksit. Leaving inconsistency using fuzzy logic. *Information and Software Technology*, 43(12):725–741, 2001.

[86] Francesco Marcelloni and Mehmet Aksit. Fuzzy logic-based object-oriented methods to reduce quantization error and contextual bias problems in software development. *Fuzzy Sets and Systems*, 145(1):57–80, 2004. Computational Intelligence in Software Engineering.

[87] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[88] D. McDermott. Planning and acting. *Cognitive Science*, 2(2):71–109, 1978.

[89] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.

[90] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[91] A.M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, february 2001.

[92] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[93] Tom Mens. On the use of graph transformations for model refactoring. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and transformational techniques in software engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer, 2006.

[94] Tom Mens, Gabriele Taentzer, and Dirk Mueller. *Model-Driven Software Development: Integrating Quality Assurance*, chapter Model-Driven Software Refactoring. IDEA Group Publishing, 2008.

[95] Tom Mens, Gabriele Taentzer, and Olga Runge. Analyzing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, September 2007.

[96] Tom Mens, Dalila Tamzalit, Michaël Hoste, and Jorge Pinna Puissant. Amélioration de la qualité de modèles: Une étude de deux approches complémentaires. *Revue des Sciences et Technologies de l'Information - Série Technique et Science Informatiques - Numéro Spécial IDM*, 29(4–5):571–599, 2010.

[97] Tom Mens and Ragnhild Van Der Straeten. Incremental resolution of model inconsistencies. In *Proceedings of the Workshop on Algebraic Description Techniques (WADT)*, volume 4409 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2007.

[98] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Proceedings of the Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer, October 2006.

[99] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. *A Framework for Managing Consistency of Evolving UML Models*, pages 1–31. Idea Group Publishing, 2005.

[100] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, 2005.

[101] Bertrand Meyer. *Object-Oriented Software Construction*, chapter 24: Using inheritance well. Prentice Hall Upper Saddle River, NJ, 2nd edition, 1997.

[102] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In *Proceedings of the International Conference Model Driven Engineering Languages and Systems (MoDELS)*, Lecture Notes in Computer Science. Springer, 2009.

[103] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - A review of experiences from applying MDE in industry. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443, Berlin, Germany, June 2008. Springer.

[104] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching, part II*, pages 285–292. Hardvard University Press, Cambridge, Massachisetts, 1959.

[105] Alix Mougenot, Xavier Blanc, and Marie-Pierre Gervais. D-praxis: A peer-to-peer collaborative model editing framework. In *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS '09, pages 16–29. Springer-Verlag, 2009.

[106] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform random generation of huge metamodel instances. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 130–145. Springer-Verlag, 2009.

[107] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[108] D. Nau, T. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F.Yaman. Shop: An HTN planning system. *Journal of Artificial Intelligence Research*, 2003.

[109] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185, 2002.

[110] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings International Conference Software Engineering (ICSE)*, pages 455–464, Washington, DC, USA, 2003. IEEE Computer Society.

[111] A. Newell and G. Ernst. The search for generality. In W. A. Kalenich, editor, *Information Processing 1965: Proceedings of IFIP Congress*, volume 1, pages 17–24, 1965.

[112] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29, April 2000.

[113] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[114] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, 2000.

[115] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification, January 2006.

[116] Object Management Group. Unified Modeling Language: Infrastructure version 2.3. formal/2010-05-03, May 2010.

[117] Object Management Group. Unified modeling language: Super structure version 2.3. formal/2010-05-05, May 2010.

[118] David L. Parnas. Software aging. In *Proceedings International Conference Software Engineering (ICSE)*, pages 279–287. IEEE Computer Society Press, May 16-21 1994. Sorento, Italy.

[119] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the International Conference Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[120] J. Scott Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 103–114, 1992.

[121] Javier Pérez. *Refactoring Planning for Design Smell Correction in Object-Oriented Software*. PhD thesis, ETSII, University of Valladolid, July 2011.

[122] Jorge Pinna Puissant, Tom Mens, and Ragnhild Van Der Straeten. Resolving model inconsistencies with automated planning. In *Proceedings of the 3rd workshop on Living with Inconsistencies in Software Development*, CEUR Workshop Proceedings. CEUR-WS.org, September 2010.

[123] Jorge Pinna Puissant, Tom Mens, and Ragnhild Van Der Straeten. Comparing automated planning approaches for model inconsistency resolution. Technical report, University of Mons, April 2011.

[124] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Automated planning for resolving model inconsistencies – a scalability study. In *MoDELS workshop on Models and Evolution*, October 2010.

[125] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Badger: A regression planner to resolve design model inconsistencies. In *Proceedings European Conference Modelling Foundations and Applications (ECMFA)*, volume 7349 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

[126] A. Pretschner and W. Prenninger. Computing refactorings of state machines. *Software and Systems Modeling*, 6(4):381–399, 2007.

[127] Ulrike Ranger and Thorsten Herme. Ensuring consistency within distributed graph transformation systems. In *Proceedings of the International Conference Fundamental Aspects of Software Engineering (FASE)*, volume 4422 of *Lecture Notes in Computer Science*, pages 368–382, 2007.

[128] Alexander Reder and Alexander Egyed. Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ASE '10, pages 347–348, New York, NY, USA, 2010. ACM.

[129] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010.

[130] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency Checking of Conceptual Models via Model Merging. In *Proceedings International Conference Requirements Engineering*, pages 221–230. IEEE Computer Society, 2007.

[131] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[132] E. D. Sacerdoti. A structure for plans and behaviour. Technical Report 109, Elsevier North-Holland, New York, 1977.

[133] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[134] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[135] Jocelyn Simmonds, Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Maintaining consistency between UML models using description logic. *Série l'objet - logiciel, base de données, réseaux*, 10(2-3):231–244, 2004.

[136] H.A. Simon and A. Newell. Heuristic problem solving: The next advance in operations research. *Operations research*, 6(1):1–10, 1958.

[137] Avik Sinha, Matthew Kaplan, Amit M. Paradkar, and Clay Williams. Requirements modeling and validation using bi-layer use case descriptions. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2008.

[138] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler, and Dana S. Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, October 2004.

[139] Jean Louis Sourrouille and Guy Caplat. Constraint checking in UML modeling. In *Proceedings of the International Conference Software Engineering and Knowledge Engineering (SEKE '02)*, pages 217–224. ACM, 2002.

[140] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World scientific, 2001.

[141] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervais. Supporting collaborative development in an open MDA environment. In *ICSM*, pages 244–253. IEEE Computer Society, September 2006.

[142] Thomas Stahl and Markus Völter. *Model Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[143] H. Tardieu, A. Rochfeld, and R. Colletti. *La méthode Merise: principes et outils*. Editions d'organisation, 1983.

[144] R. Tarjan. Depth-first search and linear graph algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory*, pages 114–121. IEEE Computer Society, 1971.

[145] Austin Tate. *Using Goal Structure to Direct Search in a Problem Solver*. PhD thesis, University of Edinburgh, 1975.

[146] Henri Theil. *Economic forecasts and policy*, volume 5 of *Contributions to economic analysis*. North-Holland Pub. Co., 1958.

[147] Alban Tiberghien, Naouel Moha, Tom Mens, and Kim Mens. Répertoire des défauts de conception. Technical Report 1303, University of Montreal, 2007.

[148] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.

[149] P. van Beek and X Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 585–590, 1999.

[150] Ragnhild Van Der Straeten. Using description logic in object-oriented software development. In *Proceedings of the International Workshop on Description Logics (DL)*, 2002.

[151] Ragnhild Van Der Straeten. *Inconsistency management in model-driven engineering: an approach using description logics*. PhD thesis, Vrije Universiteit Brussel, 2005.

[152] Ragnhild Van Der Straeten and Maja D'Hondt. Model refactorings through rule-based inconsistency resolution. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pages 1210–1217. ACM, 2006.

[153] Ragnhild Van Der Straeten, Tom Mens, and Viviane Jonckers. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, 6:139–162, June 2007.

[154] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logics to maintain consistency between UML models. In *Proceedings of the International Conference Unified Modeling Language (UML)*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2003.

[155] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in model-driven software engineering. In *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 35–47. Springer, 2009.

[156] Ragnhild Van Der Straeten, Jorge Pinna Puissant, and Tom Mens. Assessing the kodkod model finder for resolving model inconsistencies. In *Proceedings of the European Conference Modelling Foundations and Applications (ECMFA)*, volume 6698 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2011.

[157] Ragnhild Van Der Straeten, Jocelyn Simmonds, and Tom Mens. Detecting inconsistencies between UML models using description logic. In *Proceedings of the International Workshop on Description Logics (DL)*, September 2003.

[158] Arie van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In *Proceedings CSMR Workshop on Model-Driven Software Evolution (MoDSE)*, 2007.

[159] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[160] S.A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 5:246–267, 1983.

[161] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. Flexible and scalable consistency checking on product line variability

models. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ASE '10, pages 63–72, New York, NY, USA, 2010. ACM.

[162] Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. PhD thesis, Massachusetts Institute of Technology, January 1971.

[163] Dan Wu, Evren Sirin, James A. Hendler, Dana S. Nau, and Bijan Parsia. Automatic web services composition using SHOP2. In *WWW (Posters)*, 2003.

[164] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 315–324, New York, NY, USA, 2009. ACM.

[165] Chang Xu. *Inconsistency detection and resolution for context-aware pervasive computing*. PhD thesis, Hong Kong University of Science and Technology (People's Republic of China), 2008. Adviser-Cheung, S.C.

[166] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Volume II of Research and Practice in Software Engineering*, pages 199–218. Springer, 2005.