Universidade do Minho
Escola de Engenharia

Tiago Manuel da Silva Jorge

A Model Repair Application Scenario with PROVA

Outubro de 2014

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Manuel da Silva Jorge

A Model Repair Application Scenario with PROVA

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor Manuel Alcino Pereira da Cunha

Outubro de 2014

DECLARAÇÃO

Nome

Tiago Manuel da Silva Jorge

Endereço electrónico: tiagomsjorge@gmail.com Telefone: ______ / ______

Número do Bilhete de Identidade: 13800764

Título dissertação ☒ / tese ☐

A Model Repair Application Scenario with PROVA

Orientador(es):

Professor Doutor Manuel Alcino Pereira da Cunha

Ano de conclusão: 2014

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Engenharia Informática

Universidade do Minho, 31 / 10 / 2014

Assinatura: Tiago Manuel da Silva Jorge

This thesis is dedicated to my parents, Paulo and Manuela.
Thank you for your love and endless support.

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to my adviser Professor Dr. Manuel Alcino Cunha, for encouraging and mentoring my research. I would also like to thank my colleague and researcher Nuno Macedo for his priceless advice. I should also express my gratitude to José Faria for giving me the opportunity to work closely with industry, as well as for all the provided support. A great part of this work was done under financial support of grant BI2-2013_PROVA-23137_UMINHO, granted within the R&D project PROVA - Platform for Software Verification and Validation (QREN, ADI, SI 2012/023137, 2012-2014).

ABSTRACT

Model-Driven Engineering (MDE) is a well known approach to software development that promotes the use of structured specifications, referred to as models, as the primary development artifact. One of the main challenges in MDE is to deal with a wide diversity of evolving models, some of which developed and maintained in parallel. In this setting, a particular point of attention is to manage the model inconsistencies that become inevitable, since it is too easy to make contradictory design decisions and hard to recognise them. In fact, during the development process, user updates will undoubtedly produce inconsistencies which must eventually be repaired. Tool support for this task is then essential in order to automate model repair, so consistency can be easily recovered. However, one of the main challenges in this domain is that for any given set of inconsistencies, there exists an infinite number of possible ways of fixing it. While much of researchers recognise this fact, the way in which this problem should be resolved is far from being agreed upon, and methods on how to detect and fix inconsistencies vary widely. In this master dissertation a comparison between different approaches is done and an application scenario is explored in close collaboration with industry. An off-the-shelf model repair tool leveraging the power of *satisfiability* (SAT) solving is put to test, while an incremental technique of complex repair trees is implemented and evaluated as a promising, yet very distinctive competitor.

Keywords: MDE, consistency rules, model inconsistencies, model repair.

RESUMO

A *engenharia orientada aos modelos* (MDE), uma abordagem bem conhecida no desenvolvimento de software, promove a utilização de especificações estruturadas, denominadas modelos, como o artefacto primário de desenvolvimento. Um dos principais desafios neste domínio é lidar com a grande diversidade de modelos em evolução, muitas vezes desenvolvidos e mantidos em paralelo. Neste cenário é essencial gerir as inconsistências dos modelos, que se tornam inevitáveis uma vez que facilmente se tomam decisões contraditórias e de difícil reconhecimento. De facto, durante o processo de modelação, atualizações aos modelos por parte do utilizador irão sem dúvida produzir inconsistências que devem ser reparadas. Ferramentas que suportem este processo tornam-se essenciais para automatizar a reparação dos modelos, por forma a que a consistência seja facilmente recuperada. No entanto, para qualquer conjunto de inconsistências existe um número potencialmente infinito de possíveis formas de o corrigir, facto que revela ser um dos principais problemas neste domínio. Embora grande parte dos investigadores reconheça este desafio, a forma como esta problemática deve ser abordada está longe de reunir consenso e as soluções propostas variam muito. Nesta dissertação de mestrado é feita uma comparação entre diferentes abordagens à técnica de reparação de modelos, e um cenário de aplicação é explorado em estreita colaboração com a indústria. Uma ferramenta pronta a usar e que aproveita o poder do *SAT solving* é posta à prova, enquanto que uma outra técnica, incremental e baseada em complexas árvores de reparação, é implementada e avaliada como uma abordagem concorrente promissora e bastante distinta.

Palavras chave: MDE, regras de consistência, inconsistência de modelos, reparação de modelos.

CONTENTS

i PRELIMINARY STUDY 1

1 INTRODUCTION 3

1.1 Motivation 3

1.1.1 A call from industry 4

1.1.2 Problem challenges 4

1.2 Problem example 5

1.3 Contributions 6

1.4 Document structure 7

2 A SURVEY OF MODEL REPAIR TECHNIQUES 9

2.1 Heuristics-driven state-space exploration 9

2.2 Manually specifying repair information 10

2.3 Models as sequences of elementary operations 11

2.3.1 Limited state-space exploration 11

2.3.2 Automated planning 12

2.4 Relational model finding 12

2.5 Repair dependence graph 13

2.6 An incremental approach 13

2.6.1 Repair trees 14

2.7 *Least-change* repairs with Echo 14

2.8 Collection of comparison points 15

2.8.1 Main remarks 17

3 REPAIRING REQUIREMENT MODELS - A CASE STUDY 19

3.1 Application scenario 19

3.1.1 PROVA requirement models 20

3.1.2 The Boilerplates 20

3.1.3 Desired consistency rules 22

3.2 Solving approach with Echo 23

3.2.1 ECore metamodel 23

3.2.2 OCL constraints 24

3.2.3 Check and repair 26

3.2.4 Remarks 27

ii AN INCREMENTAL APPROACH WITH CHECK-AND-REPAIR TREES 31
4 MAINTAINING CHECK TREES 33
4.1 Technique overview 33
4.1.1 Tree structure and scope 34
4.1.2 Reevaluating 35
4.1.3 Lazy evaluation 36
4.2 Method implementation 36
4.2.1 Subtyping 37
4.2.2 Minimal set of logical operators 38
4.2.3 Terminal nodes and event hooks 41
4.2.4 Constraint language 42
4.3 Evaluation 43
4.3.1 Transition definition 44
4.3.2 Transition 1 - tree increment 45
4.3.3 Transition 2 - collapsing of the tree 46
4.3.4 Transition 3 - out of scope event 48
4.3.5 Transition 4 - rebuilding the tree 48
4.3.6 Formula complexity 49
4.3.7 Discussion of results 50
5 GENERATING REPAIR TREES 53
5.1 Generation mechanism 53
5.1.1 Expected logical value 53
5.1.2 Eliminating false and non-minimal repairs 54
5.2 Extending the library 55
5.2.1 Repairing each logical operator 55
5.2.2 Fixing terminal nodes 58
5.3 Repair example 59
5.3.1 Tree structure 59
5.3.2 Interpreting and simplifying repair trees 62
5.4 Challenges to the approach 64
5.4.1 Overlaps 65
5.4.2 Negative side effects 65
5.4.3 Abstract repairs 66
6 CONCLUSION 67
6.1 Main results 67
6.2 Future work 68

LIST OF FIGURES

Figure 1 Simplified UML Model of a VOD System (reprinted from Egyed (2006)) 5
Figure 2 Structural view of a requirements model 22
Figure 3 ECore metamodel 24
Figure 4 ECore metamodel annotated with OCL 26
Figure 5 Inconsistency detection 27
Figure 6 *least-change* repairs 28
Figure 7 Consistency rule for UML (taken from Reder and Egyed (2012b)) 34
Figure 8 Tree for message 'connect' 34
Figure 9 Reevaluations against changes (taken from Reder and Egyed (2012b)) 35
Figure 10 Adding consistent requirement rC 45
Figure 11 Requirement addition time for increasing scope 46
Figure 12 Adding inconsistent requirement rC 47
Figure 13 Collapse time for increasing scope 47
Figure 14 Adding out of scope requirement rD 48
Figure 15 Out of scope event time for increasing scope 49
Figure 16 Deleting inconsistent requirement rC 49
Figure 17 Rebuilding time for increasing scope 50
Figure 18 Transition times for (left to right) one, two, and five nested quantifiers 50
Figure 19 $R(a \wedge \neg(b \wedge c), t)$, where a is false, and b and c are true 55
Figure 20 Flattened repair tree 63
Figure 21 Simplified repair tree 65

LIST OF TABLES

Table 1 Comparison of approaches to model repair 16
Table 2 Repairing $a \wedge b$ 54
Table 3 Inconsistent model 59

ACRONYMS

MDE	Model-Driven Engineering
OMG	Object Management Group
MDA	Model-Driven Architecture
UML	Unified Modelling Language
OCL	Object Constraint Language
EMF	Eclipse Modelling Framework
DSML	Domain Specific Modelling Language
CSP	Constraint Satisfaction Problem
MDT	Model Development Tools
QVT	Query/View/Transformation
SAT	Satisfiability
BX	Bidirectional Transformations
SMT	Satisfiability Modulo Theories
BNF	Backus Normal Form

Part I

PRELIMINARY STUDY

1

INTRODUCTION

Model-Driven Engineering (MDE) is a well known approach to software development that promotes the use of structured specifications, referred to as models, as the primary development artifact. To support MDE, the *Object Management Group* (OMG) has launched the *Model-Driven Architecture* (MDA) initiative in 2001. As Stevens puts (Stevens, 2007), "The central idea of the OMG's Model Driven Architecture is that human intelligence should be used to develop models, not programs. Routine work should be, as far as possible, delegated to tools: the human developer's intelligence should be used to do what tools cannot."

In MDE, different models may capture different views of the same system, for instance concerning its structure and dynamics, or may represent different levels of abstraction, a common case when automating the derivation of source code implementations from high-level specifications. As an effect of the increasing adoption of MDE, large-scale industrial projects are currently being developed by hundreds of people, making use of several model instances conforming to different metamodels.

1.1 MOTIVATION

One of the main challenges in MDE is to deal with a wide diversity of evolving models, some of which developed and maintained in parallel. In this setting, a particular point of attention is to manage the model inconsistencies that become inevitable, since it is too easy to make contradictory design decisions and hard to recognise them. Providing automated mechanisms to support these evolving models becomes essential (Van Der Straeten et al., 2009), since inconsistencies have been revealed as one of the main development problems (Hessellund et al., 2007). Even a minor inconsistency can lead to serious faults in the system and be the source of project failure, so developing techniques for inconsistency management becomes crucial.

As defined by Spanoudakis and Zisman (2001), inconsistency management not only consists in the detection of inconsistencies but also in their handling. During the development process, user updates will undoubtedly produce inconsistencies which must eventually be repaired. However, the resolution of these violations is, currently, mostly a manual task. The user may manually alter the model to eliminate the violations, but this can be a very challenging problem due to the complexity of the language or the concrete model. Tool support for this task is then essential in order to automate model repair, so consistency can be easily recovered.

1.1.1 *A call from industry*

An example of the need for solutions to this problem, in this dissertation an application scenario is explored in close collaboration with industry, namely with Educed[1], a startup company created to exploit the application of formal methods for the development of safe and reliable software systems. Educed is developing the PROVA (Platform for Software Verification and Validation) platform, whose goal is to assist engineers in the development and validation of software requirements for critical systems. In PROVA, multiple views are used in the definition and analysis of a given system, each view representing a subset of relevant attributes. Complementary views include, for example, the structural, operational and behavioural view. Consistency between these views must be maintained.

1.1.2 *Problem challenges*

In the attempt to detect and repair inconsistencies various challenges arise.

One of the main challenges to model repair, as shown by Nentwich et al. (2003), is that for any given set of inconsistencies, there exists an infinite number of possible ways of fixing it. In fact, the number of alternatives may grow exponentially with the complexity of the design rule and the number of model elements involved. While much of researchers recognise this fact, the way in which this problem should be resolved is far from being agreed upon, and methods on how to detect and fix inconsistencies vary widely. Moreover, they are implemented in different programming languages, on different operating systems, use different modelling languages, and different input and output formats.

To top it off, other challenges arise such as the impact of a given identified repair, as a change that fixes one inconsistency may introduce new ones. Potential solutions should consider this problem, known as side effects, capturing dependencies between consistency rules and repair plans. Being able to tolerate the presence of certain model inconsistencies, while being able to resolve others, is also important. This may be necessary, for instance, in the early stages of software design, where design models will be incomplete and contain a lot of inconsistencies that can only be resolved in later phases of the development process.

1 http://www.educed-emb.com/

1.2 PROBLEM EXAMPLE

This section presents a simple example taken from Egyed (2006), showing that even a small model may contain several inconsistencies that must be resolved.

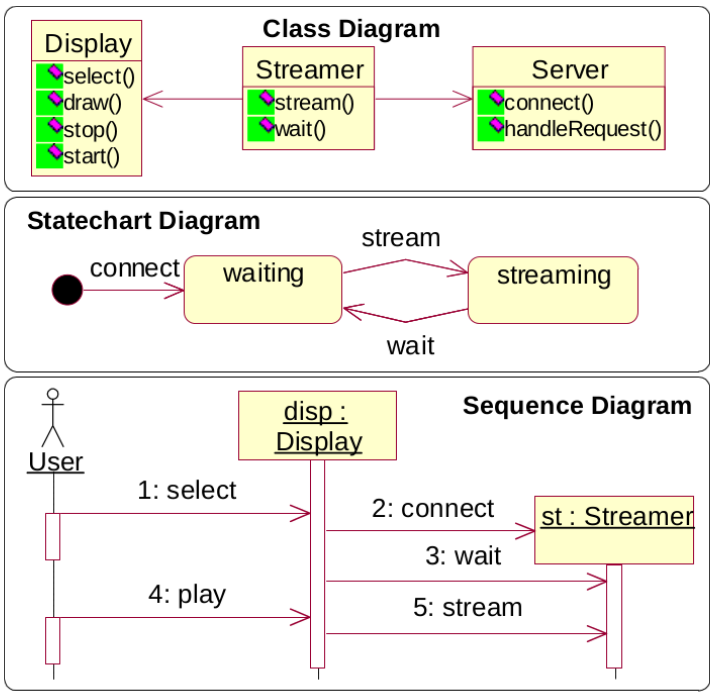


Figure 1: Simplified UML Model of a VOD System (reprinted from Egyed (2006))

Figure 1 depicts three *Unified Modelling Language* (UML) diagrams, representing a simplified *video-on-demand* (VOD) system.

The class diagram (top) captures the structure of the system, where a *Display* is used for visualising movies and receiving user input, a *Streamer* downloads and decodes movie streams, and a *Server* provides the movie data. The state-chart diagram (middle) describes the behaviour of the *Streamer* class. It first establishes a connection to the *Server* and then toggles between the *waiting* and *streaming* mode depending on whether it receives the *wait* and *stream* commands. The sequence diagram (bottom) represents the process of selecting a movie and playing it. The illustration depicts a user invoking the *select* method on an object *disp* of type *Display*. This object then creates a new object *st* of type *Streamer*, invokes *connect* and then *wait*. Finally, when the user invokes *play*, *disp* invokes *stream*.

Consistency rules describe conditions that a model must satisfy for it to be considered valid according to its metamodel. These conditions may be, for instance, syntactic well-formedness and coherence between different diagrams concerning the same model (their static semantics).

In this example, for instance, a consistency rule should state that the name of a message in the sequence diagram must match a method in the receiver's class.

This simple model contains different inconsistencies. For example, there is no *connect* method in the *Streamer* class although the *disp* object invokes *connect* on the *st* object. Besides, the *disp* object calls the *st* object even though in the class diagram only a *Streamer* may call a *Display*. Moreover, the sequence of incoming messages of the *st* object (*connect*;*wait*;*stream*) is not supported by the state-chart diagram which expects a *stream* after a *connect*.

As already mentioned, for a given inconsistency usually there are several alternative fixes, fact which represents an obstacle to existing solutions. For example, regarding the first inconsistency, a repair could create the *connect* method in the *Streamer* class, or alternatively remove its invocation in the sequence diagram.

1.3 CONTRIBUTIONS

The contributions resulting from the research activities carried out in the context of this thesis are:

- A comparison of different existing model repair techniques. In particular, a proposal is made for a collection of relevant features for comparing these methodologies. This collection should mature and develop into a well-defined taxonomy.

- Exploration of an application scenario in close collaboration with industry, namely with Educed. As a first attempt to find a solution to a provided case study, Echo (Macedo et al., 2013) was used, an off-the-shelf model repair tool that automates inconsistency detection and repair using a solver based engine. Some remarks are made about the pros and cons experienced throughout the design. This analysis allowed one to assess the suitability of this tool to be used as a model repair component in Educed's PROVA platform, also serving as user feedback for future Echo developments.

- Implementation, evaluation and critical analysis of an incremental approach to model repair (Reder and Egyed (2012b), Reder and Egyed (2012a)), as an alternative to Echo. Its incremental behaviour focusing on quick feedback was a driving factor for the interest in this method, in view of the fact that the front-end component of the PROVA platform intensively explores direct user interaction for the modelling activity. A Python library was implemented, an open source tool that is free to test and customise. This tool was the outcome of an opportunity to discover implementation challenges and test the technique applicability to an arbitrary modelling and constraint language. Different procedures of the technique are tested, their performance analysed, and the overall results discussed. A strategy for interpreting and simplifying repairs is suggested, and some challenges to the approach are discussed.

1.4 DOCUMENT STRUCTURE

This dissertation is structured as follows.

Part i introduces the preliminary study carried out in the context of this thesis. In Chapter 2, different existing model repair techniques will be discussed from Section 2.1 to 2.7, after which, in Section 2.8, a proposal is made for a taxonomy of relevant features for comparing check and repair methodologies. In Chapter 3, Section 3.1 starts by introducing PROVA and its modelling language as a case study. Some examples of requirements are given and the desired constraints on their static semantics are defined. Following this, in Section 3.2 the introduced requirements and rules are modelled and tested in Echo, an off-the-shelf model repair tool. Some remarks are made about the pros and cons experienced throughout the design.

In Part ii, an incremental approach to model repair is tackled, as an alternative to Echo. In Chapter 4, the incremental verification method presented in Reder and Egyed (2012b) is carefully explained, implemented and tested. Section 4.1 gives an overview of the consistency checking mechanism. Section 4.2 describes the implemented Python library applying the checking behaviour. After this, in Section 4.3, the tool is tested for constraints on a metamodel of the PROVA's front-end. In particular, different tree transformations and constraints are tested, their performance analysed, and the overall results discussed. In Chapter 5, the repair method introduced in Reder and Egyed (2012a) is presented as an extension to the previous checking mechanism. Section 5.1 presents an overview of this repair technique, while, in Section 5.2, the library is extended with a repair method. In Section 5.3, this new functionality is tested, results are discussed, and a strategy for interpreting and simplifying repair trees is suggested. Finally, in Section 5.4, challenges to this repair approach are discussed. Chapter 6 concludes this dissertation, where Section 6.1 summarises the main results obtained throughout this work, and Section 6.2 proposes some goals as future work.

2

A SURVEY OF MODEL REPAIR TECHNIQUES

As already pointed out, one of the main challenges to model repair is that for any given set of inconsistencies, there exists an infinite number of possible ways of fixing it. Puissant et al. (2013) believe that search-based approaches such as meta-heuristics, could be applied to this problem. They argue the problem 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. On the other hand, Reder and Egyed (2012a) advocate against fully automatic approaches, such as heuristics that replace the role of the human designer in repairing inconsistencies. According to this authors, repairing models should be an activity that goes hand in hand with the creative process of modelling. Indeed, methods on how to detect and repair inconsistencies vary widely. Moreover, they are implemented in different programming languages, on different operating systems, use different modelling languages, and different input and output formats.

Taking this diversity of approaches as motivation, in this chapter different existing model repair techniques will be discussed from Section 2.1 to 2.7, after which, in Section 2.8, a proposal is made for a collection of relevant features for comparing check and repair methodologies. This collection is used to compare some of the studied approaches. As a future goal, this collection should mature and develop into a well-defined taxonomy. Main remarks are presented in Subsection 2.8.1.

2.1 HEURISTICS-DRIVEN STATE-SPACE EXPLORATION

Hegedus et al. (2011) present an approach that is based on graph transformations which generates quick fixes for *Domain Specific Modelling Languages* (DSMLs). According to the authors, while domain-specific editors are usually capable of ensuring that elementary editing operations preserve syntactic correctness (by e.g. syntax-driven editing), most DSMLs include additional language-specific consistency rules that must also be checked. Their aim is thus to provide a domain-independent framework (that is applicable for a wide range of DSMLs), which can efficiently compute complex fixing action sequences even when multiple, overlapping inconsistency rule violations are present in the model. Thereby, the authors propose to adapt the concept of quick fixes (also called error correction, code completion) found in the programming languages domain to DSMLs.

To capture inconsistency rules of a DSML, they use graph patterns that define declarative structural constraints. Their technique uses graph transformation rules to specify elementary fix operations (policies). These operations are automatically combined by a structural constraint solving algorithm that relies on heuristics-driven state-space exploration to find quick fix sequences efficiently. The technique guarantees that the number of inconsistencies on the model decreases, even if side-effects occur. This is achieved by applying every candidate fix to the inconsistent model and detecting and counting the inconsistencies in the resulting model. The heuristics-guided algorithm automatically selects the best solutions. The engine is capable of generating solutions for a given local scope independently of the total number of violations in other parts of the model.

2.2 manually specifying repair information

Xiong et al. (2009) present an approach that combines the detection of errors with the generation of actions to repair them on UML models. They use their own language to define the consistency relations. This language, called Beanbag, has a syntax similar to *Object Constraint Language* (OCL) and together with the specification of consistency rules, it allows to specify how models breaking such rules should be fixed. Rules can then be run either in checking or fixing mode to repair models. More precisely, Beanbag attaches fixing actions to primitive constraints and functions, and composes them through logic operators and other high-level constructs.

This approach is completely automatic, i.e., without requiring user interaction. In contrast to this method, there are approaches which generate fixing actions purely from a consistency relation, but require human interventions in executing the actions, by specifying some locations to fix, choosing one among a set of actions or filling some missed parameters. Nevertheless, the authors believe both types of approaches are important to consistency management, because while some consistency relations are suitable to be established all the time through automatic fixing, some are suitable to be manually resolved by humans.

As the authors state, compared to *Constraint Satisfaction Problem* (CSP) approaches, Beanbag is a more lightweight approach in the sense that it requires users to describe the fixing behaviour in the Beanbag program, and thus does not suffer from the scalability problem. Moreover, they compare Beanbag to heuristics-driven methods claiming their approach provides a more clear, predictable fixing semantics, so that end users can clearly know how their updates affect other parts of the model.

2.3 MODELS AS SEQUENCES OF ELEMENTARY OPERATIONS

Blanc et al. (2008) propose to represent models by sequences of elementary construction operations, rather than by the set of model elements they contain. Structural and methodological consistency rules can then be expressed uniformly as logical constraints on such sequences.

The authors argue that structural and methodological inconsistency detection should be supported uniformly. Methodological rules constrain the overall construction process, whereas structural rules only constrain the model obtained at the end of this process. Regarding the model life cycle, those two kinds of rules are complementary as structural rules constrain model states whereas methodological rules constrain model changes. They stress the fact that, to their knowledge, no existing approach allows to define methodological consistency rules. For instance, OCL is mainly used to specify model structural inconsistency rules, not being possible to define methodological constraints.

The authors point out that approaches based on elementary operations are the most suitable for large-scale distributed environments, where there is a need to merge parallel changes that have been performed to models in a distributed way.

The approach was validated by building a Prolog engine, Praxis, that detects violations of structural and methodological constraints specified on UML models and requirement models. It is worthwhile to note that all checks are executed in batch mode. The entire model is loaded in memory, and the rules are verified one after the other on the entire model. An incremental consistency checking would be much more effective. Nevertheless, the authors claim to be able to compute how rules access the sequence of elementary operations using the introspection facilities of Prolog, making it possible to generate an incremental checker for a set of rules.

2.3.1 *Limited state-space exploration*

Silva et al. (2010) developed an approach that generates repair plans based on Praxis. The generated repair plans also consist of sequences of Praxis actions, in this case those that are needed to resolve as many inconsistencies as possible causing as few new inconsistencies as possible. Their key contribution is a search algorithm that is optimised to find the shortest plan that fixes the bigger number of inconsistencies, by exploring a limited and configurable subset of all possible plans. In fact, as there exists an infinite number of ways to resolve inconsistencies, this approach has a configurable exploration level which reduces the number of repairs. Nevertheless, this solution involves the danger of not being able to resolve the inconsistencies.

The algorithm is also fitted to build repair plans that start by fixing the most recent causes of inconsistencies. Moreover, besides obtaining repair plans that correct all inconsistencies in a model, it is possible to get partial plans that start by proposing fixes to the inconsistencies that were more recently introduced in the model. The authors argue that this capacity helps the developer when correcting models that have too much inconsistency and which would therefore require too much time to compute a complete repair plan.

2.3.2 *Automated planning*

Puissant et al. (2013) implemented Badger, an automated regression planner also built with Praxis that generates model inconsistency resolutions, given concrete and previously detected inconsistencies (the tool requires as input a model and a set of inconsistencies). The automated planning is a logic-based approach originating from artificial intelligence. The planner does not require the user to specify resolution rules manually or to specify information about the causes of the inconsistency.

As the authors state, to make the approach useful in practice, the resolution preferred by the user should be among the first generated resolution plans. For this purpose, the user can adapt the order in which resolution plans are presented by assigning different costs to edit operations. Furthermore, entire resolution plans can be omitted by attaching an infinite weight to certain actions.

Although the approach has only been stress-tested on a Java system and on UML models, the authors claim that relying on a metamodel independent representation (using sequences of elementary model operations) makes it straightforward to apply the method to other types of models as well.

2.4 RELATIONAL MODEL FINDING

Van Der Straeten et al. (2011) assessed the viability of using Kodkod (Torlak and Jackson, 2007) to perform model repair. Kodkod is a SAT-based constraint solver for first order logic with relations, transitive closure, bit-vector arithmetic, and partial models. Kodkod is used as a model finder used in a wide range of applications, including, for instance, code checking, test-case generation, and lightweight analysis of Alloy (Jackson, 2012) models. Kodkod implements relational bounds, i.e., relational variables of any arity are bounded by sets of tuples. Basically, the upper bound specifies the tuples that a relation may contain, while the lower bound specifies the tuples that it must contain.

In this approach, a repaired model is found by relaxing the bounds on some entities and associations. More specifically, nodes and edges suspected of causing the inconsistencies are removed from the lower-bound, and the upper-bound is augmented to allow additions. To minimise repairs, they first use an external procedure to identify such potentially guilty model elements. However, this technique does not ensure minimality of the repairs, it only handles one inconsistency at a time, and is still not fully automatised (e.g., the relaxation of upper-bounds is performed manually). The authors claim that, performance wise, Kodkod is not viable for model repair of large size models.

2.5 REPAIR DEPENDENCE GRAPH

Demsky and Rinard (2005) present a model-based approach to data structure repair, a technique for enabling programs to execute successfully in the presence of otherwise fatal data structure corruption errors. The method involves two views: a concrete view of the data structures as they are represented in the memory and an abstract view that models the data structures as sets of objects and relations between objects.

A set of model definition rules, encapsulating the data structure representation complexity, translates the concrete data structures to the sets and relations in the abstract model. The key consistency constraints are then expressed using the sets and relations in this model, defining important data structure consistency properties. An automatically generated repair algorithm finds and repairs any data structures that violate the defined consistency properties. These violations are repaired by automatically translating model repairs back through the model definition rules to automatically derive a set of data structure updates that implement the repair. The compiler uses goal-directed reasoning to statically map these model repair actions to data structure updates.

A repair dependence graph is used to capture dependencies between consistency constraints, repair actions, and the abstract model. Supporting formal reasoning about the effect of repairs on both the model and the data structures, this graph presents a set of conditions that identify a class of cycles whose absence guarantees that all repairs will successfully terminate. An algorithm then removes nodes in the graph to eliminate problematic cycles, preventing the repair procedure from choosing repair strategies that may not terminate.

2.6 AN INCREMENTAL APPROACH

Egyed (2006) presents an incremental approach for quickly, correctly, and automatically deciding what consistency rules to evaluate when a model changes. Consistency rules are treated as black-box entities and a form of profiling is used to observe their behaviour during evaluation in order to identify what model elements they access. The set of model elements that affects the truth value of a consistency rule is referred to as its *change impact scope*. This rule needs to be reevaluated if and only if one such model element changes, a necessary condition for an efficient incremental approach. Egyed demonstrates that his method produces complete and small (albeit not minimal) scopes.

While inconsistencies are detected quickly, following a spirit of toleration, engineers are not required to fix them right away. All known inconsistencies are continuously tracked while engineers explore them according to their interests in the model. This is a non-trivial problem because the scope of an inconsistency is continuously affected by model changes. In fact, it is significant to understand that the approach maintains a separate scope of model elements for every instance of consistency rule. This scope is computed automatically during evaluation and used to determine when to reevaluate rules. In the case of an inconsistency, this scope tells the engineer all the model elements that were involved.

2.6.1 *Repair trees*

Egyed (2007) and Egyed et al. (2008) present how to repair inconsistencies in models and how the generated choices are evaluated. However, according to the authors this approach is overly conservative and generates repairs for all model elements accessed by the validation of an inconsistency, while often only a subset thereof causes the inconsistency. In contrast, by eliminating false and non-minimal repairs, the method introduced in Reder and Egyed (2012a) vastly reduces the number of repair alternatives (however, not necessarily the exponential growth inherent to the problem).

This method combines the syntactic and dynamic structure of an inconsistent design rule to pinpoint exactly which parts of the inconsistency must be repaired. As a result, it generates a tree of repair actions. A repair tree organises the repair actions in a hierarchical manner that reflects the structure of the design rule. The authors claim that, for this reason, the repair tree is intuitive to understand and grows linearly with design rule complexity. However, these repair plans only consist of sequences of abstract edit operations, which the user must manually instantiate in order to repair the models. Thus, this is not a fully automatic repair, as it requires human intervention.

The authors believe that the approach is applicable to arbitrary modelling and constraint languages.

2.7 *least-change* REPAIRS WITH ECHO

Macedo et al. (2013) present Echo, a tool based on model finding for model repair and model transformations. Echo is deployed as an Eclipse plugin, developed on top of the popular *Eclipse Modelling Framework* (EMF), with metamodels being specified in ECore. To enhance metamodels with constraints, it follows the technique proposed by *Model Development Tools* (MDT), of embedding OCL constraints as metamodel annotations. Moreover, Echo also supports the specification of *Query/View/Transformation-Relations* (QVT-R OMG (2011)) *Bidirectional Transformations* (BX). QVT-R is a declarative language designed to specify constraints between related models. BX can be seen as a generic approach to recover inter-model consistency, being a particular case of model repair. Actually, if no propagation direction is imposed and both models are encompassed in a single global artifact, the problems become equivalent. Thus, Echo is able to check and repair both intra-model and inter-model consistency.

Its engine works by translating both metamodels (annotated with OCL) and QVT-R transformations to Alloy, a lightweight formal specification language with support for automatic model finding via SAT solving. Besides being correct, in the sense that resulting models are always fully consistent, Echo is also minimal, as it follows the principle of *least-change*. This principle states that repaired models should be as close as possible to the original inconsistent ones. There may be more than one consistent model at minimal distance, so Echo presents all possible repaired models by increasing distance, allowing the user to choose the desired one, at which time the update is effectively applied to the original model.

Since models can be seen as graphs, the standard graph edit distance, that just counts additions and deletions of nodes and edges, can be used to measure distances. Echo automatically infers this measure for any given metamodel. A user-parametrised model distance measure is also available, that requires the user to specify the allowed edit operations in the metamodel, enabling a finer degree of control over valid repairs.

2.8 COLLECTION OF COMPARISON POINTS

After studying different approaches, a collection of the most relevant check and repair features was made. Those which appear to reveal greater importance in comparing the different methodologies are listed in Table 1. Due to some ambiguity faced and the need for a more detailed study in the future, a record of such features is currently only made for a subset of the mentioned approaches. Also, for those considered, some answers are yet to be discovered, and are left blank in the table.

Several features were chosen concerning incrementality, user interaction, automation, completeness, manual effort, inconsistencies resolution, side effects, *least-change* behaviour, control over repairs and underlying technique. Being a direct result of the study carried out, this collection represents a proposal for a vector of comparison between different model repair approaches:

- Incremental - an approach is incremental if it does not operate in batch mode, allowing reusing results from previous checking phases. Here, only the necessary set of rules engaged in the last model modification need to be tested. This allows the approach to scale.

	incremental	user can choose repairs	repairs automatically applied	complete	no manually specified repairs	one inconsistency at a time	side effects handling	principle of *least-change*	control over repair generation	syntactic-based
Xiong et al. (2009)			✓	✗	✗			✗	✓	✓
Puissant et al. (2013)		✓	✗	✓	✓	✗		✗	✓	✗
Van Der Straeten et al. (2011)	✗		✓	✓	✓	✓	✓	✗	✗	✗
Reder and Egyed (2012a)	✓	✓	✗	✓	✓	✓	✓	✗	✗	✓
Macedo et al. (2013)	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗
Demsky and Rinard (2005)	✗		✓		✓	✗	✓	✗	✓	✓

Table 1: Comparison of approaches to model repair

- User can choose repairs - more control is given to the user when he/she can choose between the generated repairs. This flexibility is important in the modelling activity.
- Repairs automatically applied - repair plans may consist of sequences of abstract edit operations, which the user must manually instantiate in order to effectively repair the models. This is not a fully automatic repair, as it requires human intervention in executing the actions, by specifying some locations to fix, choosing one among a set of actions or filling some missed parameters.
- Complete - all possible repairs are considered, albeit in a bounded exploration space. Non complete approaches may fail to repair inconsistencies or to include the one the designer wants.
- No manually specified repairs - the user is not required to manually specify repair operations along with the consistency rules, a laborious and error-prone activity with no guarantee for completeness or correctness. Do not confuse this with user intervention in instantiating repairs, occurring only after some abstract repair have been generated.
- One inconsistency at a time - generated repairs only concern a certain detected inconsistency (and eventually its side effects). This approach is more scalable than resolving all consistencies at once. Besides, this method follows the spirit of tolerating inconsistencies.
- Side effects handling - when applying a generated repair other inconsistencies are typically raised, and mechanisms that trace and resolve this dependencies should be im-

plemented. This can also be solved by approaches like solving, since they resolve all consistencies at the same time and ensure the generated models are correct.

- Principle of *least-change* - the principle of *least-change* requires repaired models to be as close as possible to the original. This approach becomes more predictable to the designer.

- Control over repairs generation - most of the repairs are undesirable. Allowing to somehow steer the repair generation mechanism, can result both in a more predictable fixing semantics and in the elimination of undesirable fixes. This can be achieved, for instance, by defining specific allowed repair operations and assigning weights to them.

- Syntactic-based - these techniques typically derive repair plans by syntactic analysis of the constraints and model instances. This is usually very efficient and scalable, but can be less expressive and flexible than, for instance, solving. Moreover, syntactic analysis is not as well suited to deal with multiple inconsistencies, nor inconsistencies that affect a large portion of the model.

2.8.1 *Main remarks*

Although the approach from Xiong et al. (2009) repairs models automatically, the user has to manually specify the fixing behaviour, an error-prone approach without guarantees of completeness. In Puissant et al. (2013), once the resolution plan is generated, the user must replace temporary elements by concrete elements, so this is not a fully automatic approach. However, no fixing annotations need to be manually specified, and users can adapt the order in which resolution plans are presented by assigning different costs to edit operations.

In Demsky and Rinard (2005), a repair dependence graph is used to capture dependencies and resolve side effects. However, it is not incremental nor evaluates models without the need to translate them to an abstract representation.

Among all approaches, the one from Reder and Egyed (2012a) appears to be the one which scales better, since it is iterative, evaluates models directly, does not necessarily repair all inconsistencies at once, and follows a syntactic-based analysis. However, the approach is not fully automatic as the user must instantiate proposed repairs, does not follow the *least-change* principle and no control over repairs generation is given.

Although less scalable when compared to Egyed's approach, the one from Macedo et al. (2013) is fully automatic, generating repaired models which are as close as possible to the original, and provides control over repairs generation through the specification of allowed edit operations. Straeten's technique (Van Der Straeten et al., 2011) does not ensure minimality of the repairs nor control over its generation.

3 REPAIRING REQUIREMENT MODELS - A CASE STUDY

An application scenario was explored in close collaboration with industry, namely with Educed, a startup company created to exploit the application of formal methods for the development of safe and reliable software systems. Educed is developing the PROVA platform, whose goal is to assist engineers in the development and validation of software requirements for critical systems. Section 3.1 starts by introducing PROVA and its modelling language. Some examples of requirements are given and desired constraints on their static semantics are defined.

As a first attempt to find a solution to this case study, an off-the-shelf model repair tool was used to implement it. Therefore, in Section 3.2 the introduced requirements and rules are modelled and tested in Echo (Macedo et al., 2013), a tool that automates inconsistency detection and repair using a solver based engine. Echo was chosen since it was the tool which revealed the most active support and clearer documentation. Furthermore, it was fairly easy to install and test. It is deployed as an Eclipse plugin, developed on top of the EMF, and supports constraints specified in the standard OCL. Some remarks are then made about the pros and cons experienced throughout the design.

The chapter ends by concluding that, currently, Echo may not be well suited to the requirements of the PROVA platform. This served as a motivation for studying and developing the incremental approach presented in Part ii of this dissertation. Nevertheless, the performed modelling and test activities served as a realistic case study from industry which may help developing Echo.

3.1 APPLICATION SCENARIO

This section starts by introducing PROVA, in particular the motivation behind its requirements modelling framework. Next, some structural boilerplates (requirement specification blocks) are presented and examples of requirements (their instances) are given in the context of an illustrative model. Finally, some desired consistency rules are informally defined, constraints concerning the static semantics of the language of boilerplates.

3.1.1 *PROVA requirement models*

Developing a critical software system is recognisably an arduous task, both by the usual inherent complexity of the problem to solve and by the very high assurance required. In this process, getting the requirements right is key, as it is against them that the final system behaviour is analysed and verified. Undetected errors at the requirements stage propagate throughout the development and later become much more costly to correct.

Nowadays, requirement analysis is usually conducted in a relatively informal way, since requirements are commonly described in natural language using a text document. Although this process can follow a rigorous methodology, because requirements are written in natural language it becomes impossible to perform an automatic and rigorous analysis. Moreover, as these documents are usually pretty long, carrying out a proper analysis without any automated support is a daunting task. Modelling languages like UML are very expressive and already form part of the curriculum of a software engineer. However, most UML tools available in the market focus on model construction only and not on their analysis or simulation. This is due to the semi-formal nature and semantics of UML, which hinders the development of verification tools. On the other hand, various languages and tools for formal modelling have been developed in the academia context, for instance Alloy (Jackson, 2012). While these offer opportunities for automated analysis, the specification languages provided are not attractive enough to most engineers.

PROVA explores the design of a requirement specification language that is familiar to the engineer, but also suitable for formal and automated processing. This language comprises a set of requirement specification blocks named boilerplates (Hull et al., 2011). For each boilerplate a sentence in natural language expressing its semantics is defined. Each sentence has a set of placeholders to be filled by the engineer, this being the way in which actual requirements are instantiated. A requirements model is then translated to a Satisfiability Modulo Theories (SMT) problem providing the desired simulation capabilities, consistency analysis and verification of properties.

In PROVA, multiple views are used in the definition and analysis of a given system, each view representing a subset of relevant attributes defined with boilerplates of a specific type. Complementary views include, for example, the structural, operational and behavioural view.

3.1.2 *The Boilerplates*

PROVA supports different types of boilerplates, one for each view of the system being modelled. For instance, structural boilerplates define the static part of the system, i.e., existing entities, their attributes and relationships; operational boilerplates express operations carried out by the different entities on the system; behavioural boilerplates define the system dynamics as a state machine. As a proof of concept, we will focus on structural boilerplates only. These can be further subdivided in subtypes such as *Specialisation*, *Relation*, and *Restriction*. In the following examples, while the first two boilerplates introduce entities and relations to a specification, the third one restricts existing relations. Figure 2 shows the structural view (in UML-like notation, to be more familiar) of a requirements model where these examples appear, so it helps understanding the context in which they are used.

ENUMERATION This is a boilerplate of type *Specialisation*. As its type suggests, it is used to define a requirement specialising a certain model entity into an enumeration of more specific entities. Below, its formal syntax in Backus–Naur Form (BNF), and a requirement example along its informal semantics:

```
<enumeration> ::= "every" <entity> "shall be enumerated by" <entity-list>
<entity-list> ::= <entity> "and" <entity-list> |  <entity>
<entity> ::= string
```

Example: *every State shall be enumerated by Empty and NonEmpty*
Informal semantics: An entity of type *State* is always an entity of type *Empty* or *NonEmpty*.

ASSOCIATION This is a boilerplate of type *Relation*. It is used to define a named association relating two model entities and respecting some multiplicity. Below, its formal syntax in BNF, and a requirement example along its informal semantics:

```
<association> ::= "every" <entity> "shall have" <mult> <relation> <entity>
<mult> ::= "Some" | "One" | "Lone" | "Any"
<relation> ::= string
```

Example: *every Message shall have One message_source Partition*
Informal semantics: *message_source* relates to every *Message* exactly *One Partition*, its source. This implicitly defines a transposed relation *_message_source* with a predefined multiplicity.

GENERALISATION This is a boilerplate of type *Restriction*, being used to constrain composing relations. Below, its formal syntax in BNF, and a requirement example along its informal semantics:

```
<Generalisation> ::= "every" <relation-list> "of" <entity> "is" <relation-list>
                     "of" <entity>
<relation-list> ::= <relation> "of" <relation-list> |  <relation>
```

Example: *every _message of Message is _channel_state of NonEmpty*

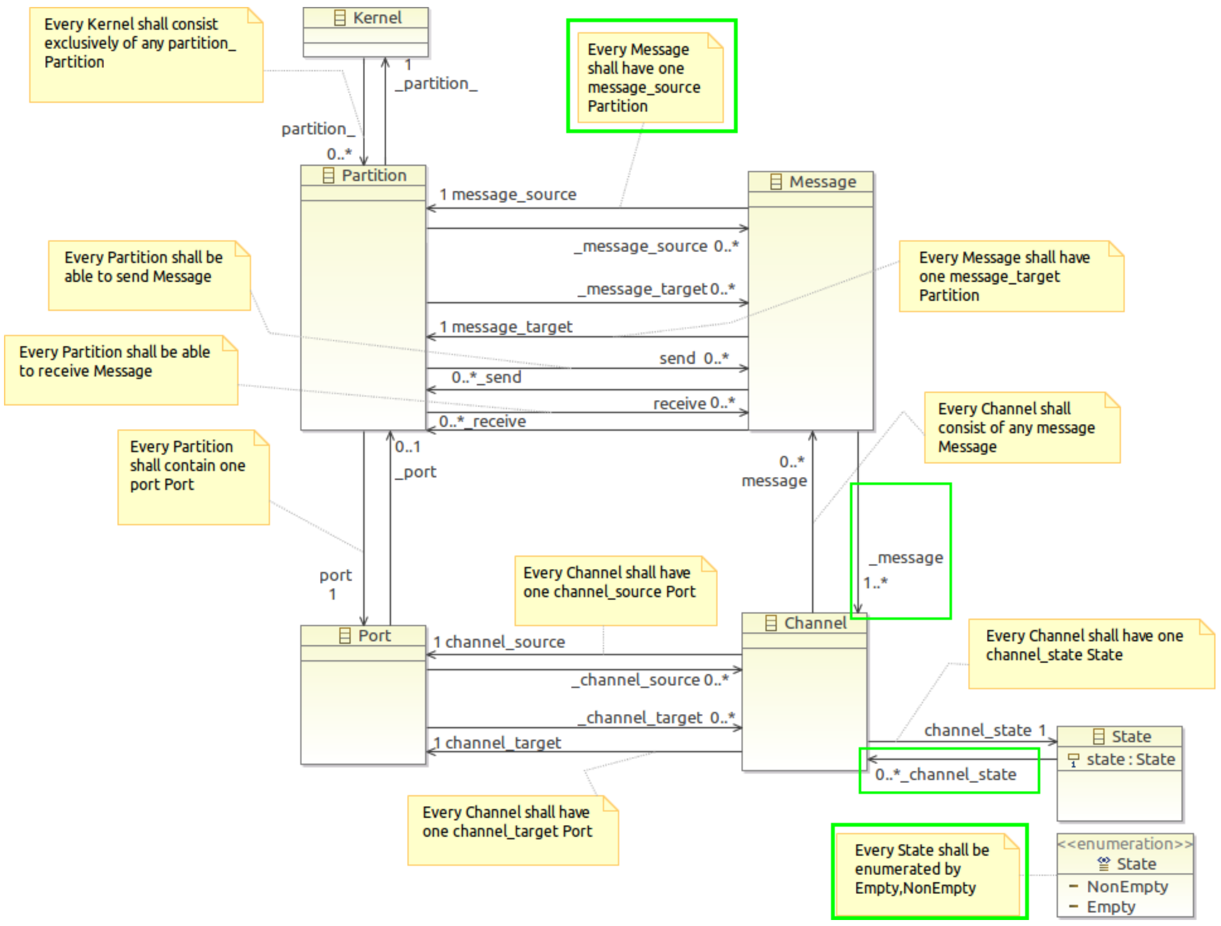

Every Kernel shall consist exclusively of any partition_ Partition
Kernel
1
partition
partition_
0..*
Every Message shall have one message_source Partition
Partition
Message
1 message_source
_message_source 0..*
_message_target 0..*
1 message_target
send 0..*
0..*_send
receive 0..*
0..*_receive
Every Partition shall be able to send Message
Every Partition shall be able to receive Message
Every Message shall have one message_target Partition
Every Channel shall consist of any message Message
Every Partition shall contain one port Port
0..1
_port
0..*
message
_message
1..*
port
1
Every Channel shall have one channel_source Port
Port
Channel
1 channel_source
_channel_source 0..*
_channel_target 0..*
1 channel_target
Every Channel shall have one channel_state State
channel_state 1
0..*_channel_state
State
state : State
Every Channel shall have one channel_target Port
Every State shall be enumerated by Empty,NonEmpty
<<enumeration>>
State
NonEmpty
Empty

Figure 2: Structural view of a requirements model

Informal semantics: The *channel_state* of a channel associated with a *Message* via *message* is *NonEmpty*.

3.1.3 *Desired consistency rules*

For a system's model to be valid for further analysis, the static semantics of the Boilerplate's language must be obeyed first, being either intra-view or inter-view constraints. These constraints on the static semantics of the language can not be captured by a simple grammar.

As examples of consistency rules that should be obeyed, consider the following (informally defined) three constraints:

- Rule *SPEC_ONCE* - An entity can only be specialised once, unless no requirement `Enumeration` is used. For example, having 2 requirements: *every A shall be enumerated by A1 and A2 and A3* and *every A shall be enumerated by A4 and A5* is illegal.

- Rule *NO_SELF_SPEC* - Entities do not specialise themselves. Circular or self-references between any requirements of type *Specialisation* are illegal. For instance, having: *every A shall be enumerated by A1 and A2 and A3* and *every A2 shall be enumerated by A4 and A* is illegal. This rule together with the previous one ensure that specialisations remain consistent.

- Rule *WELL_TYP_COMP* - The lists of composing relations in requirements of type *Restriction* must be well-typed, i.e., the associated requirements of type *Relation* must share some entity types in a way such that hooking them becomes possible. For instance if we have *... r2 of r1 ...* (equivalent to *r2∘r1*) in one of the lists of a requirement of type `Generalisation`, then two requirements of type `Relation` must exist, such that the type of the second entity of *r1* is the same as the type of the first entity of *r2*.

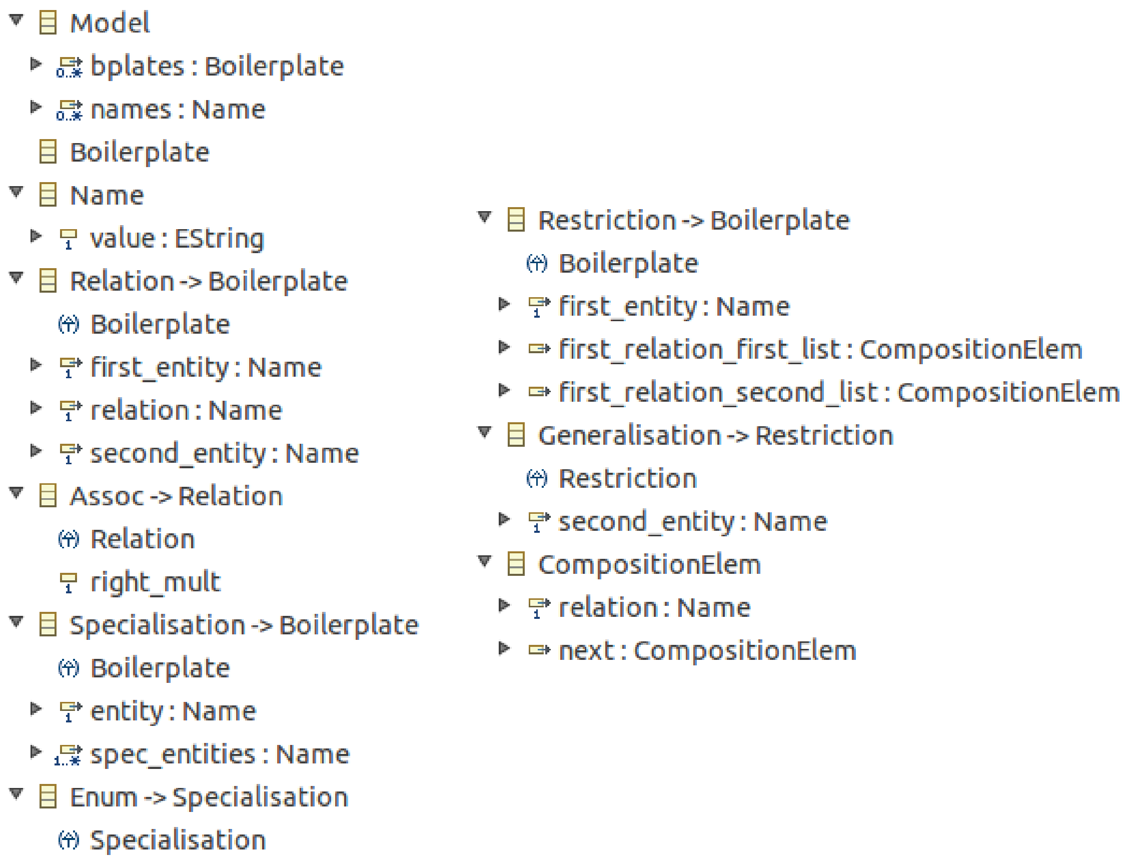
Besides employing a consistency checker that would perform the verification of these rules, it would also be desirable for PROVA to provide feedback, by warning the user of the existing conflicts and providing suggestions for correction.

3.2 SOLVING APPROACH WITH ECHO

In this section, the boilerplates presented in Subsection 3.1.2 are modelled using ECore, the language for specifying metamodels in Echo, and the consistency rules informally defined in Subsection 3.1.3 are formally specified as embedded OCL annotations. The check and repair mechanisms are tested, and some remarks are made about the pros and cons experienced throughout the design.

3.2.1 *ECore metamodel*

In Echo, metamodels are defined using the ECore language. The boilerplates presented in Subsection 3.1.2 were modelled using this language. Figure 3 shows the resulting metamodel, capturing the variable attributes of the boilerplates (the relevant ones), that is, their placeholders, as collections of names (class `Name`). Classes `Relation`, `Specialisation` and `Restriction` host common attributes required by their subclasses. Here, one subclass per type is shown, respectively `Assoc`, `Enum` and `Generalisation`, the boilerplates previously introduced. Since order is relevant for modelling lists of composing relations, a self-referenced class is used, namely `CompositionElem`. This is due to a current limitation of the tool and not of ECore, which already covers sequences.



```
Model
  bplates : Boilerplate
  names : Name
  Boilerplate
Name
  value : EString
Relation -> Boilerplate
  Boilerplate
  first_entity : Name
  relation : Name
  second_entity : Name
Assoc -> Relation
  Relation
  right_mult
Specialisation -> Boilerplate
  Boilerplate
  entity : Name
  spec_entities : Name
Enum -> Specialisation
  Specialisation
Restriction -> Boilerplate
  Boilerplate
  first_entity : Name
  first_relation_first_list : CompositionElem
  first_relation_second_list : CompositionElem
Generalisation -> Restriction
  Restriction
  second_entity : Name
CompositionElem
  relation : Name
  next : CompositionElem
```

Figure 3: ECore metamodel

3.2.2 *OCL constraints*

To enhance the Ecore metamodels with additional constraints, Echo follows the technique of embedding OCL rules as annotations. Next, the consistency rules informally defined in Subsection 3.1.3 are formally specified as embedded OCL annotations. While the first rule was quite simple to specify, the other two introduced a considerable degree of complexity.

Rule *SPEC_ONCE* is defined for each requirement of type `Enum`, so its context becomes that same class (line 1):

```
1 <Enum>
2  spec_once:
3   Specialisation.allInstances()
4   ->forAll(s | s <> self implies s.entity <> self.entity)
```

As the entity of this kind of requirement can only be specialised once, the rule must prevent the existence of any other `Specialisation`, `s`, sharing the same entity (lines 3 and 4, `<>` denoting the difference operator).

Rule *NO_SELF_SPEC* is a reachability property, so it was necessary to add the extra attribute `Name.specs` (a class self-reference) to the metamodel, use auxiliary constraints (`name_specs`, lines 5 and 11), and resort to semantically more complex OCL operators (`closure` and `includes`):

```
 1 <Name>
 2  no_self_spec:
 3   not self->closure(specs)->includes(self)
 4
 5  name_specs:
 6   self.specs->forAll(n|
 7     Specialisation.allInstances()->exists(spec|
 8       spec.entity = self and spec.spec_entities->includes(n)))
 9
10 <Specialisation>
11  name_specs:
12   self.spec_entities->forAll(e|self.entity.specs->includes(e))
```

Auxiliary constraint `name_specs` is a two-part rule ensuring that the new attribute `Name.specs` exclusively (lines 5 to 8) references every specialisation of entity `Name` (lines 11 and 12). Resorting to the `closure` operation applied on the new attribute, `no_self_spec` then ensures that no `Name` (entity) specialises itself, by prohibiting its own inclusion in the closure (line 3).

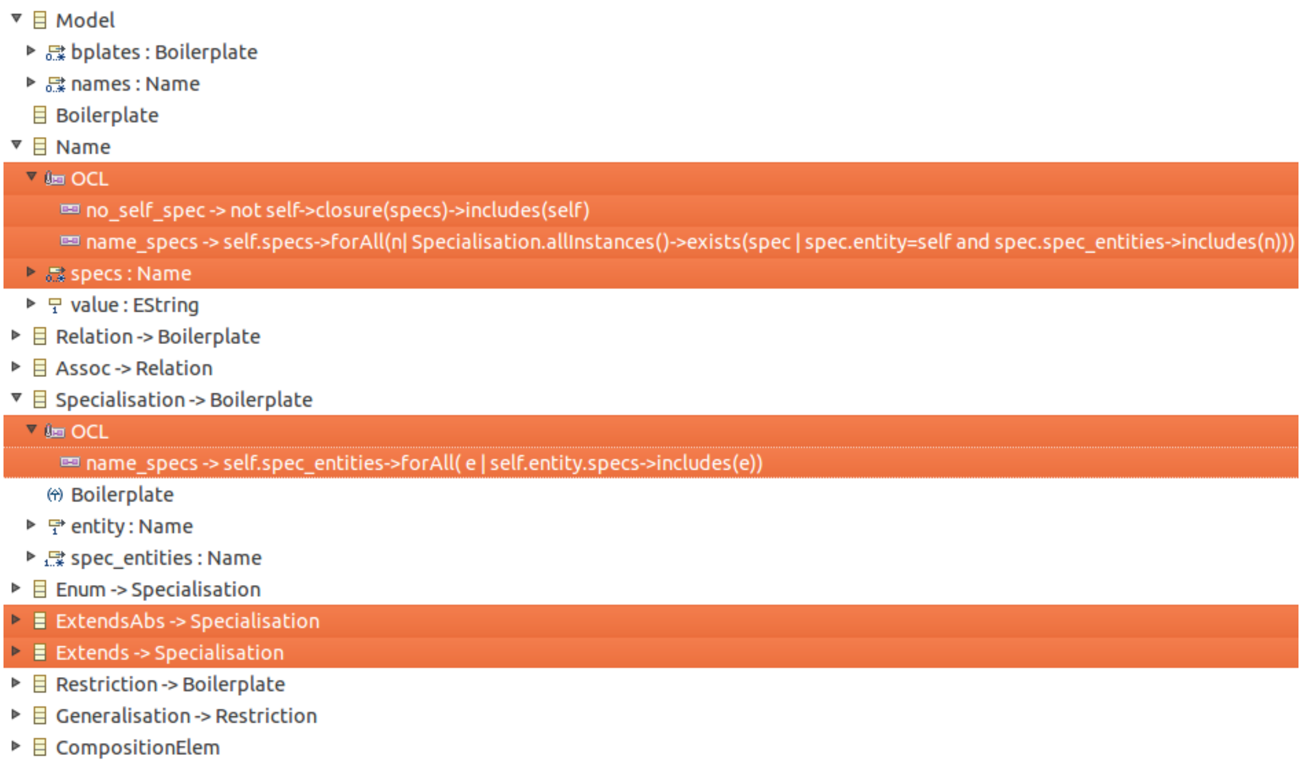
Rule *WELL_TYP_COMP* is defined for every requirement of type *Restriction*, ensuring that each composition of two relations is well-typed. Only the constraint for the first list is shown, since this is analogous for the second. Also having a considerable degree of complexity, this rule resorts to nested quantifiers of multiple variables, and to semantically rich operations such as `union` and `closure`:

```
<Restriction>
 well-typ-comp:
  self.first_relation_first_list
  ->union(self.first_relation_first_list->closure(next))
  ->forAll(e1,e2|
     e1.next=e2 implies Relation.allInstances()
     ->exists(a1,a2|
        e1.relation = a1.relation and
        e2.relation = a2.relation and
        a1.second_entity = a2.first_entity))
```

As already mentioned, when order became necessary, as in the case of these lists, a self-referenced class was used, namely `CompositionElem` (the type of `first_relation_first_list`), where each instance references a `relation`. All pairs `e1,e2` of such elements are fetched from the `union` of the very first element with the `closure` applied on its attribute `next` (lines 3 to 5). From here, all pairs which compose are required to be consistent with two requirements `a1,a2` of type `Relation` ('a' from subtype `Association`), which must exist (lines 6 and 7). To be consistent, their relation names must match and `a1` must hook to `a2` by obeying `a1.second_entity = a2.first_entity` (lines 8 to 10).

3.2.3 *Check and repair*

Figure 4 shows part of the ECore metamodel, now annotated with OCL rules. Rule `no_self_spec` is highlighted, as well as the extra attribute and auxiliary constraints needed. Two subclasses of `Specialisation` were added for testing purposes and are highlighted also, namely `ExtendsAbs` and `Extends`.

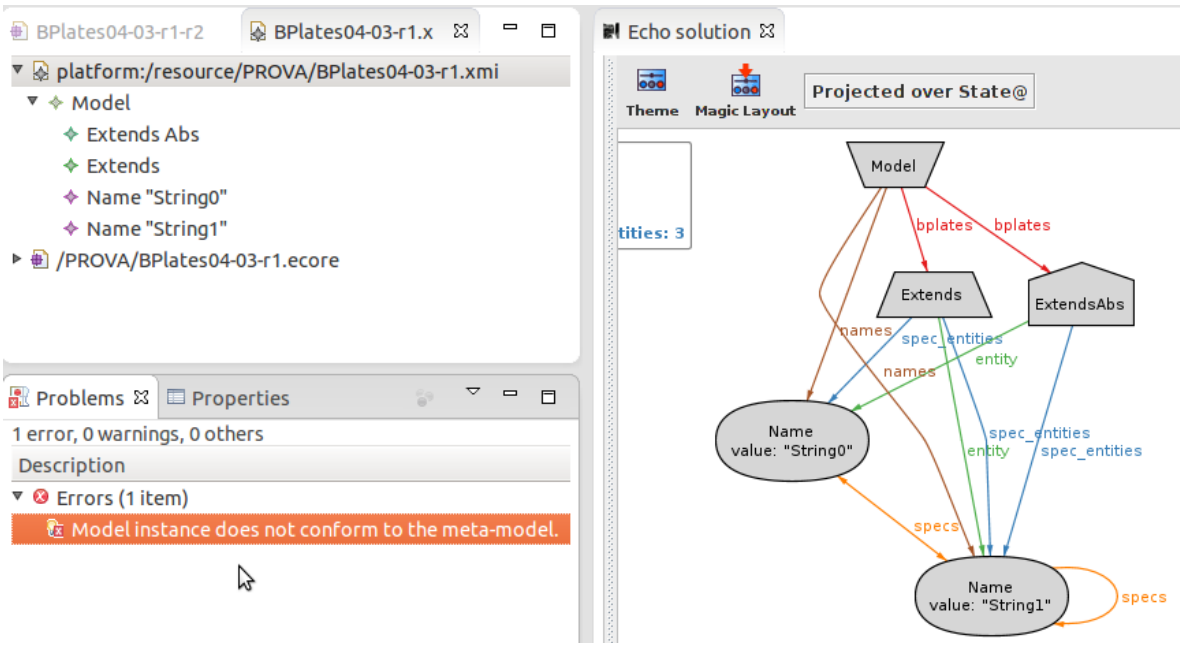


```
▼ Model
   ▶ bplates : Boilerplate
   ▶ names : Name
  Boilerplate
▼ Name
   ▼ OCL
       no_self_spec -> not self->closure(specs)->includes(self)
       name_specs -> self.specs->forAll(n| Specialisation.allInstances()->exists(spec | spec.entity=self and spec.spec_entities->includes(n)))
   ▶ specs : Name
   ▶ value : EString
▶ Relation -> Boilerplate
▶ Assoc -> Relation
▼ Specialisation -> Boilerplate
   ▼ OCL
       name_specs -> self.spec_entities->forAll( e | self.entity.specs->includes(e))
     Boilerplate
   ▶ entity : Name
   ▶ spec_entities : Name
▶ Enum -> Specialisation
▶ ExtendsAbs -> Specialisation
▶ Extends -> Specialisation
▶ Restriction -> Boilerplate
▶ Generalisation -> Restriction
▶ CompositionElem
```

Figure 4: ECore metamodel annotated with OCL

Resorting to SAT solving, Echo is able to detect instances of the metamodel violating specified constraints. Figure 5 shows a small instance which violates rule `no_self_spec`, since there are specialisation cycles between entities `String0` and `String1`. As shown, Echo correctly reported this design inconsistency.

Besides detecting inconsistencies, Echo is able to repair erroneous instances (also by SAT solving) fully automatically. The repair mechanism follows the principle of *least-change*, in


BPlates04-03-r1-r2
BPlates04-03-r1.x
platform:/resource/PROVA/BPlates04-03-r1.xmi
Model
Extends Abs
Extends
Name "String0"
Name "String1"
/PROVA/BPlates04-03-r1.ecore

Problems
Properties
1 error, 0 warnings, 0 others
Description
Errors (1 item)
Model instance does not conform to the meta-model.

Figure 5: Inconsistency detection

that it generates a valid model which is as close as possible to the original. The measure of distance (or difference) is calculated based either on the number of elementary modifications made to the model as a graph (Graph Edit Distance mode), or on the number of applied custom operations, optionally defined by the user (Operation Based Distance mode). Alternative repairs are generated by request, in which case successively greater distances are explored. Figure 6 shows two possible repairs (GED mode) generated for the previous inconsistency.

3.2.4 *Remarks*

INTUITIVE WITH GOOD OCL SUPPORT Echo was quite intuitive to use, in that every check and repair feature was easily found and applied. The tool proved to have a good support for OCL, as every consistency rule was successfully tested, even those requiring semantically complex operations such as closures.

AUTOMATIC BUT UNDESIRED REPAIRS Echo computed repaired models fully automatically, i.e., without any user intervention. Besides, no manually specified repair information was needed during the modelling process. The tool was always correct, in the sense that every resulting model was always fully consistent with no side effects. However, the desired repairs did not always arise among the first models generated by Echo. For instance, instead of deleting one of the requirements, as shown in Figure 6, one could simply want to change the

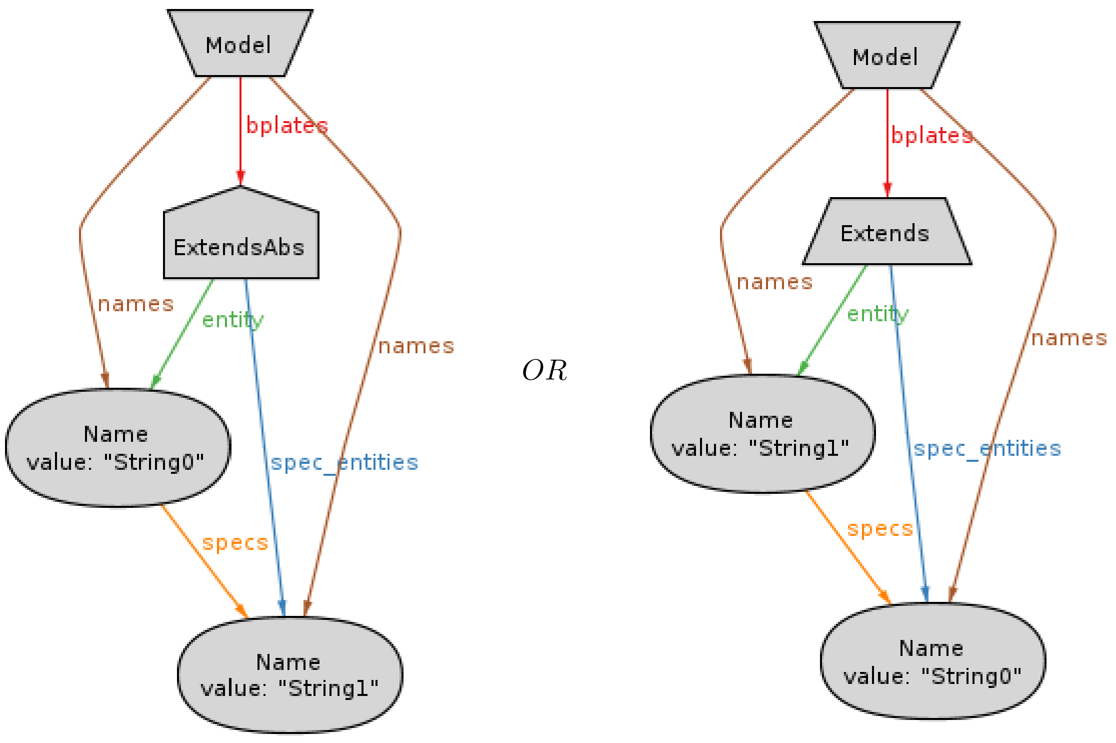


Figure 6: *least-change* repairs

entity of requirement `Extends` from `String1` to `String2`. In fact, the *least-change* principle does not always yields the desired repairs firstly.

NOT FAST ENOUGH In every test run, Echo revealed reasonable response times. In particular, checking instances for consistency took execution times on the order of tenths of a second, while generating repair suggestions required a longer processing, yet never above a few seconds. However, the tested model instances have a relatively small size when compared to full requirement models, so these performance times are not satisfactory enough for an approach with scalability issues such as Echo, as it is based on SAT solving.

LACKING INCREMENTALITY AND TRACEABILITY The key component of Echo is a target oriented model finding procedure built on top of a SAT solver. For this reason, before anything else, models have to be translated into a proper evaluation representation. This not only constitutes an overhead in performance but hinders the implementation of an incremental mechanism. Consequently, Echo does not reuse results from previous checks, always having to consider the whole set of design rules, regardless of the impact that the last modification may have had. This represents an obstacle to the scalability of the approach. Another problem experienced was the inability of the tool to pinpoint which particular rules were violated by the last modification. Here, resorting to the extraction of UNSAT cores can be a solution to consider. These problems make

VERDICT The identified problems related to automation, performance, incrementality, and traceability, led to the conclusion that, currently, Echo may not be well suited to be integrated into the PROVA platform as a model repair component. In fact, a more quick feedback is required in a highly interactive environment, pinpointing exactly which rules were violated, thus making clearer to the user how to fix the models, not having to rely on automatically generated suggestions. This served as a motivation for studying and developing the incremental approach presented in Part ii.

Part II

AN INCREMENTAL APPROACH WITH CHECK-AND-REPAIR TREES

4

MAINTAINING CHECK TREES

As one of the two approaches chosen to be studied in the context of this thesis, in this chapter the incremental verification method presented in Reder and Egyed (2012b) is carefully explained, implemented and tested. Aiming at a quick, correct, and automatic decision on what specific consistency rules to evaluate when a given model change happens, this technique represents a quite different, yet promising alternative approach to solving. Its incremental behaviour focusing on quick feedback was a driving factor for the interest in this method, in view of the fact that the front-end component of the PROVA platform intensively explores direct user interaction for the modelling activity.

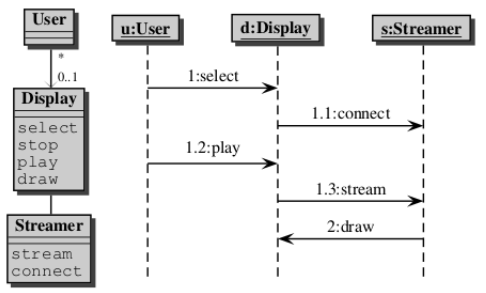
Section 4.1 gives an overview of the consistency checking mechanism, explaining the structure of a tree, its scope, reevaluation and lazy behaviour. Thereafter, Section 4.2 describes the implemented Python library applying the checking behaviour, a tool free to test and customise which was the outcome of an opportunity to discover implementation challenges and test the technique applicability to an arbitrary modelling and constraint language. Besides detecting inconsistencies, this method can be extended to generate repair suggestions, as introduced in Reder and Egyed (2012a). Later in Chapter 5 this repair functionality will be presented as an extension (also implemented) to the library. In Section 4.3 the tool is tested for constraints on a metamodel of the PROVA's front-end, the application component which would greatly benefit from incremental verification due to its interactive nature. In particular, different tree transformations and constraints are tested, their performance analysed, and the overall results discussed.

4.1 TECHNIQUE OVERVIEW

This section gives an overview of the consistency checking mechanism. Starting with and example of how the structure of a tree relates to its consistency rule and current scope, it then describes how reevaluations are performed in a bottom-up fashion, ending by introducing the crucial concept of lazy evaluation.

4.1.1 *Tree structure and scope*

As an example of how this method works, consider the consistency rule shown in Figure 7, defined over UML models. This rule states that (first operand of the conjunction) every two objects $l1, l2$, related by some message m in the sequence diagram, must be associated in the class diagram, through an attribute a, according to m's direction. Moreover (second operand), it declares that each message must correspond to a defined operation in the class of every receiver object.



$$
\begin{array}{l}
\mathit{Message}\ m:\\
\quad (\exists l_1 \in m.\mathit{receiveEvent.covered},\\
\quad\ \ l_2 \in m.\mathit{sendEvent.covered}:\\
\quad \exists a \in l_2.\mathit{represents.type.ownedAttribute}:\\
\quad a \neq \mathit{null} \Rightarrow a.\mathit{type} = l_1.\mathit{represents.type})\\
\qquad\qquad\qquad\qquad \wedge\\
\quad (\forall l \in m.\mathit{receiveEvent.covered}:\\
\quad \exists o \in l.\mathit{represents.type.ownedOperation}:\\
\quad o.\mathit{name} = m.\mathit{name})
\end{array}
$$

Figure 7: Consistency rule for UML (taken from Reder and Egyed (2012b))

For each message m, a validation tree (as shown in Figure 8 for message *connect*) is created and dynamically maintained at run-time. Each tree follows the syntactic structure of the consistency rule, each node corresponding to a logical operator and storing the logical value of the subtree (or subformula) having it as its root. The node truth value is computed by recursively validating its child nodes.

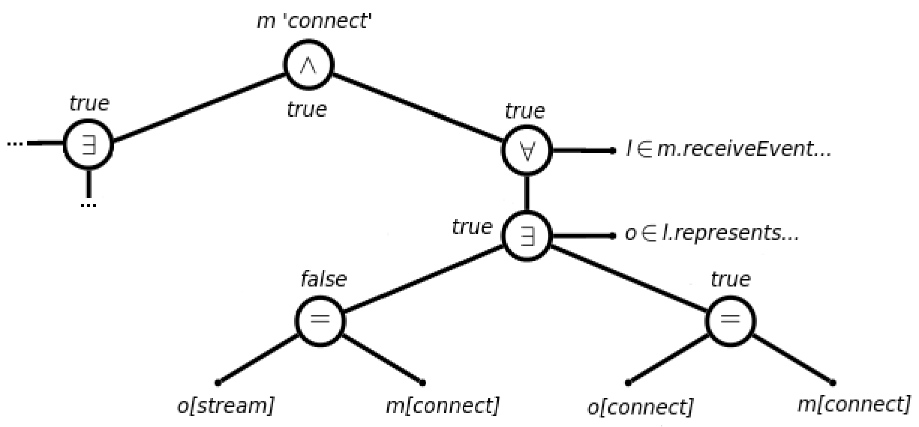


Figure 8: Tree for message 'connect'

For each element referenced by the range of a quantifier, a subtree (or branch) representing the condition of the quantifier must be created. For instance, the range of the universal quantifier *m.receiveEvent.covered* references every l object life line which is receiving message m in the sequence diagram. In the figure one can see that this universal quantifier has only one subtree, meaning that only one life line is receiving the *connect* message. In turn, the class associated with this lifeline, *Streamer*, has two o operations, *stream* and *connect*, which are compared with the message in two separate subtrees. The leafs of a tree reference elements/properties of the model. Examples are $o[stream]$ and $m[connect]$ which represent the names being compared. An element can be seen as a property of another element, so this two names are used interchangeably.

The set of properties accessed through these leaves represents the scope of the tree, necessary for getting the states of the model required to evaluate the tree consistency. The size of this scope is kept to a minimum for lazy evaluation purposes, as we shall see in Subsection 4.1.3.

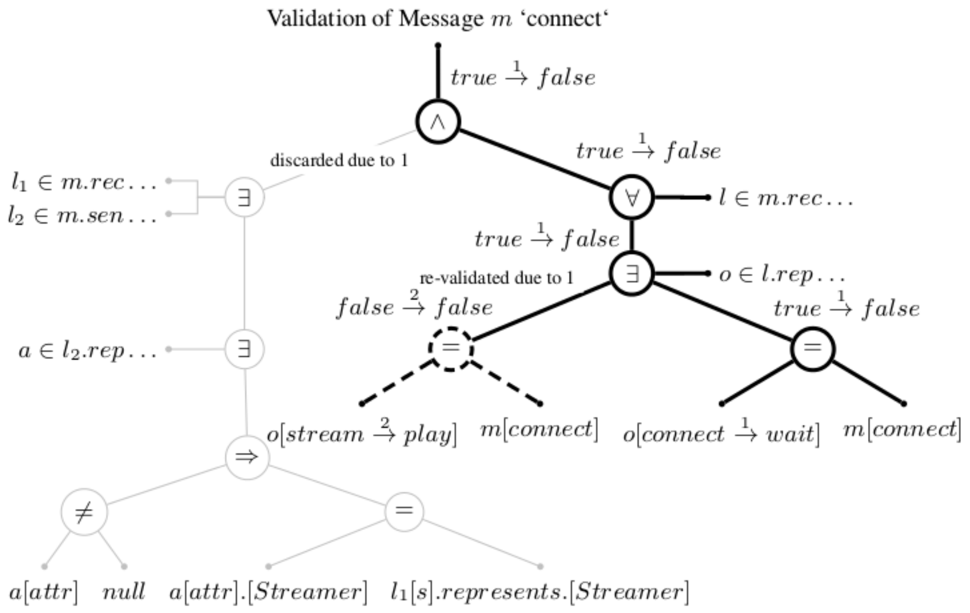

Validation of Message m 'connect'
$true \xrightarrow{1} false$
discarded due to 1
$true \xrightarrow{1} false$
$l_1 \in m.rec\ldots$
$l_2 \in m.sen\ldots$
$l \in m.rec\ldots$
$true \xrightarrow{1} false$
re-validated due to 1
$o \in l.rep\ldots$
$false \xrightarrow{2} false$
$true \xrightarrow{1} false$
$a \in l_2.rep\ldots$
$o[stream \xrightarrow{2} play]$
$m[connect]$
$o[connect \xrightarrow{1} wait]$
$m[connect]$
$a[attr]$
$null$
$a[attr].[Streamer]$
$l_1[s].represents.[Streamer]$

Figure 9: Reevaluations against changes (taken from Reder and Egyed (2012b))

4.1.2 *Reevaluating*

Changes to the model, being element additions, element deletions or changes to element properties, should be monitored in order to trigger tree reevaluations, otherwise these would become obsolete. In fact, if a change to the model occurs, then all those trees need reevaluation which include the changed property in their scope. This approach is incremental since, once a validation tree has been created, any subsequent change to the model will result in a possible update of the tree state (reevaluation), where previously computed information is reused.

As the accessed model elements (scopes) are referenced at the bottom of the tree (the leaves), the impact of a model change always start at the leaves and is propagated upward towards the root. Therefore, while the initial generation and validation of a tree are strictly top down, incremental reevaluations are, instead, bottom up.

Figure 9 shows two reevaluations (as results of two modifications), one represented by a thick line (labelled 1) and the other by a dashed line (labelled 2). The first change (1) is the renaming of the operation `connect` to `wait`. The second change (2) is the renaming of operation `stream` to `play`. While reevaluation 1 propagates upwards until it reaches the root of the tree (thus changing its consistency as a whole), reevaluation 2 does not propagate to upper nodes, since the equality remains false, despite the modification. This demonstrates that changes propagate upward for as long as the nodes are affected by the change only. Since the consistency tree stores all intermediate Boolean results, only changes to these results must be computed anew.

4.1.3 *Lazy evaluation*

As a result of reevaluation 1, the left branch of the conjunction at the top is marked as discarded. In fact, whenever one of the two branches of a conjunction is evaluated to false, the other branch can be discarded, as its logical value will not change that of the conjunction. Only when the former turns true, it becomes necessary monitoring and evaluating the latter again. This reasoning can also be applied to quantifiers. For example, as an existential quantifier is equivalent to a concatenated disjunction, it follows that if the existential quantifier evaluates to true, then only one of its consistent elements must be monitored, as the quantifier can change its truth value only if (at least) that element validation fails.

This mechanism keeps the size of the scope to a minimum in the sense that if any minimal set of elements sufficient to determine the formula truth value is found, no other element will need evaluation (thus unnecessary to monitor) until some state change occurs which renders that set insufficient. This minimum scope and its management correspond to the implementation of the concept of lazy evaluation since, by definition, lazyness states that evaluations should only be made when needed, ie when some result is required. Thanks to this lazy evaluation approach, a reevaluation focuses on the filtered validation tree only and ignores changes that would have affected discarded parts of the validation tree. This process saves both memory (as trees become smaller) and has the potential to improve performance because a change becomes less likely to affect the validation tree. This is further discussed in Section 4.3.

4.2 METHOD IMPLEMENTATION

A Python library applying the incremental verification method described in the previous section was implemented. This task was carried out since the current tool that supports this approach (the Model/Analyser) is deployed as a plug-in for he IBM Rational Software Architect framework, which is proprietary software. Moreover, besides ending up with a tool free to test and customise, it was an opportunity to discover and better understand the implementation challenges in detail. By independently implementing a library it was possible to freely define a constraint language (simpler than OCL, the one the plugin uses) and evaluate the approach more flexibly. As for the modelling language, Python itself was a suitable choice where classes and objects were used, respectively, to create metamodels (statically) and models (dynamically). As claimed by the authors in Reder and Egyed (2012b), the approach is designed to be applicable to arbitrary modelling and constraint languages, so this was also an opportunity to corroborate that.

This library is tested in Section 4.3 for constraints on a metamodel of the PROVA's front-end. This test case was chosen in order to test the approach suitability in a realistic scenario. Moreover, the front-end is the application component which would greatly benefit from incremental verification due to its interactive nature. This option also justifies the choice for a dynamic language such as Python, as the front-end is fully coded in JavaScript (not chosen instead for familiarity reasons).

The following implementation description refers only to the consistency checking mechanism. Later in Chapter 5 we will present the repair functionality as an extension to this library.

4.2.1 *Subtyping*

Subtyping is a kind of polymorphism wherein a name may denote instances of many different classes as long as they are related by some common superclass. In object-oriented programming, which was the programming model followed, this is often referred to simply as polymorphism. This technique became useful since the different logical operators were implemented as classes sharing method names, method signatures (and even some method implementations), and attributes, while at the same time requiring different implementations for some of their methods. For instance, while the method `reevaluate` is the same in every situation, the method `evaluate` is specific for each individual logical operator. Thus, a superclass `Formula` was implemented for hosting common data and behaviour.

Method `reevaluate`, an example of a shared procedure, is triggered by some change to the model, calls `evaluate` for rechecking consistency, and recursively propagates itself to upper nodes, stopping at the root or when the new logical value of the current node equals its previous state (reevaluation 2 from Figure 9 is an example of the latter case):

```
class Formula:
    def reevaluate():
        evaluate()
        if oldResult != result and parent != None:
            parent.reevaluate()
```

4.2.2 *Minimal set of logical operators*

Each logical operator corresponds to a different subclass of `Formula`. The set of implemented subclasses form a minimal set of first order logic. This way, by combining the logical operators in this set, one is capable of expressing any formula. The supported operators are `Negation` for negation, `Equals` for equality, `Conjunction` for conjunction, and `Exists` for the existential quantifier. From this minimal set other operators can be derived, such as disjunction, implication and the universal quantifier.

The choice for a minimal set of operators was made for simplicity and fast prototyping. However, the size of a tree becomes larger as more operators are eventually needed for expressing the same formula. This naturally implies not only more memory usage but some potential performance loss, as the number of procedure calls for traversing a tree grows. Therefore, implementing each desired logical operator separately should be a more efficient version for future implementation.

Next we will see the implementation of method `evaluate` for each operator, where the logical value of a node is calculated based on its child nodes.

Whenever a node is evaluated, meaning that one wants to check the consistency of the tree having that node as its root, the new logical value is stored in the instance variable `result`. For this reason, and because the `reevaluate` method needs to compare this fresh value with its previous state, the first thing to do is to save `result` in `oldResult` before updating it.

Validating a `Negation` is very simple, as it suffices to evaluate its single subtree and negate that result. Notice, however, that this validation of the child node may not be necessary if it is already evaluated. This happens in reevaluations, where nodes call their parents only after being checked:

```
class Negation(Formula):
    def evaluate():
        oldResult <- result
        evaluate childFormula if not evaluated
        result <- not childFormula.result
```

Validating `Equals` is also straightforward, being done by testing the equality of the values accessed through its two children, instances of class `Access` corresponding to leaves of the tree:

```
class Equals(Formula):
    def evaluate():
        oldResult <- result
        result <- (leftAccess.value == rightAccess.value)
```

Validating a conjunction is more complex, mainly due to the implementation of the lazy evaluation concept, where one of the two child nodes may be deactivated (or discarded):

```
 1 class Conjunction(Formula):
 2     def evaluate():
 3         oldResult <- result
 4
 5         if leftChildFormula is active:
 6             evaluate leftChildFormula if not evaluated
 7             if leftChildFormula.result == true:
 8                 activate rightChildFormula if not active
 9             else:
10                 result <- false
11                 deactivate rightChildFormula
12                 return
13
14         if rightChildFormula is active:
15             evaluate rightChildFormula if not evaluated
16             if rightChildFormula.result == true:
17                 if leftChildFormula is not active:
18                     activate leftChildFormula
19                     evaluate()
20                 else:
21                     result <- true
22                     return
23             else:
24                 result <- false
25                 deactivate leftChildFormula
26                 return
```

After recursively validating one active child (lines 6 and 15), in the case its result is false, the conjunction is also false, so its sibling can be deactivated (lines 11 and 25). This sibling can already be inactive, in which case the request is ignored. If the former is otherwise true, its sibling needs, instead, to be activated (lines 8 and 18) and evaluated (again, lines 6 and 15), because its logical value will dictate the result of the conjunction. Notice that every validation is only performed if the concerned child has not already been evaluated. The same happens for activation purposes.

The implementation for `Exists` is also a bit complex:

```
 1 class Exists(Formula):
 2     def evaluate():
 3         oldResult <- result
 4 
 5         if formula is None:
 6             result <- false
 7         else:
 8             evaluate formula if not evaluated
 9             result <- formula.result
10             if oldResult == false and result == true:
11                 trueBranch <- get first true branch
12                 for every other branch:
13                     deactivate branch
14                 formula <- trueBranch
15                 return
16 
17         if oldResult == true and result == false:
18             reset formula
19             build formula
20             evaluate()
```

This type of node have a variable number of child branches, each one being relative to an element included in the range of the quantification. These branches are organised as a concatenated disjunction referenced by `formula`. If this formula is empty it means that there are no elements to evaluate, so the result must be false (lines 5 and 6). This situation can occur if there are no model elements included in the range of the quantifier, or simply because `formula` was previously referencing only one element satisfying the quantifier which was removed (better understood next). If this is not the case, `formula` is evaluated (if necessary) and its result is assigned to that of the quantifier (lines 8 and 9). Following this, two important cases must be handled in order to respect the lazy evaluation mechanism. One corresponds to the quantifier turning true, the other one being the inverse case (lines 10 and 17, respectively). In the former situation, since at least one element which satisfies the quantification now exists (the first found is chosen), any other elements can be discarded and the branch of such element assigned to `formula` (lines 11 to 15). In the latter case (possibly also reached when `formula` is empty), as the element which satisfied the quantifier, now does not, it becomes necessary to search for other satisfaction evidence. To this end, it is necessary to rebuild `formula` (by adding branches for the elements in the range of the quantifier) and rerun the algorithm (lines 18 to 20). Handling these two cases may be quite expensive, as discussed in Section 4.3.

4.2.3 *Terminal nodes and event hooks*

In order to reevaluate a consistency tree in response to a model change, a mechanism must be implemented which associates these changes (events) with actions to execute on the tree. These associations are done with instances of `EventHook`:

```
class EventHook():
    def add(handler):
        handlers += handler
    def trigger(args):
        for handler in handlers:
            handler(args)
    def clearHandlers(leaf):
        remove from handlers every handler belonging to leaf
```

Each object in a scope should maintain attributes of type `EventHook`, each representing an event type associated with a set of actions to execute (`handlers`) whenever that specific event is triggered (method `trigger`) on the object. There are three kinds of events, namely changes, additions and deletions. Handlers are methods in terminal nodes which are responsible for updating the tree and start reevaluations. It is important to note that these associations should be eliminated (method `clearHandlers`) when objects are removed from the scope of the tree, so that their events no longer trigger calls to previously associated handlers.

One should better understand this mechanism by analysing an example for class `Access`, representative of a leaf node. This class is not a formula but rather a node that references elements and properties of the model. Its parent is always an `Equals` (which accesses model information through its children) and it has a single handler called `changed`:

```
class Access():
    def changed(ref, att):
        find ref, att in access_list
        update every subsequent reference
        for every newRef:
            newRef.onChange += changed
        parent.reevaluate()
```

The most important attribute of this class is the list of references to the observed elements/properties, namely `access_list`. For example, in the expression `a.b.att` one intends to access the value of property `att` of an element `b`, which in turn is a property of an element `a`. In this case, the list would keep references to `a`, `b`, and `att`. As an handler, method `changed` is invoked through the event management mechanism whenever any observed element (`a` or `b`) or the final attribute (`att`) changes (event of type change). This method first updates `access_list` so that this always keeps the correct path to the final attribute (line 4). Besides keeping the correct path, the associations between events and handlers should also be updated. To this end, in lines 5 and 6, for each new referenced object in the path, method `changed` itself is being added as an handler to `onChange`, the object's `EventHook` of type change. After these updates, a reevaluation is started (line 7). Thus, it is through this event mechanism that changes in models lead to reevaluations.

Class `Source`, also a terminal node, may be seen as a structure analogous to `Access`, but in which two more handlers are necessary, `added` and `deleted` (events of type addition and deletion), invoked when an element is added or removed from the range of some quantifier, respectively:

```
 1 class Source():
 2     def added(ref):
 3         if parent.result == false:
 4             parent.addBranch(ref)
 5             parent.reevaluate()
 6
 7     def deleted(ref):
 8         parent.deleteBranch(ref)
 9         if parent.result == true:
10             parent.reevaluate()
```

These two handlers invoke the associated `Exists` object (the parent), which updates the tree by adding or removing branches (lines 4 and 8). Also here we can see the application of lazy evaluation. If the existential quantifier is true, adding a new branch will not change it to false, so the addition and reevaluation is skipped. In the other hand, if the quantifier is false, removing some branch will not change it to true, so no reevaluation is needed.

4.2.4 *Constraint language*

In order to define consistency rules a simple language was implemented. The parsing of a formula results in a syntax tree from which a corresponding consistency tree is generated, already with all the necessary leaf handlers associated to the events of the model elements included in its starting scope. As already stated, certain logical operators (such as the universal quantifier) are translated into combinations of operators from the minimum set supported.

As an example of a consistency rule defined using the implemented language, consider constraint `enumerated_once` (lines 21 to 31) on the PROVA's front-end, which metamodel was statically designed with Python classes (here partially presented):

```
class Monitored():
    def init():
        onChange = EventHook()
        ...
#### Metamodel ####
class Boilerplate(Monitored):
    def init(ty, sbty, ...):
        type <- ty
        subtype <- sbty
        ...
class Requirement(Monitored):
    def init(blpt, ent, ...):
        bplate <- blpt #Boilerplate
        entity <- ent
        ...
class FtEndModel(Monitored):
    def init():
        requirements <- [] # [Requirement]
        ...
        #constraints
        enumerated_once <- "
                            forall r1 in requirements : (
                                r1.bplate.subtype = 'ENUM'
                                ->
                                forall r2 in requirements : (
                                    r2.bplate.type = 'EXTENDS'
                                    ->
                                    ( r1 != r2 -> r1.entity != r2.entity )
                                )
                            )
                           "
```

This consistency rule states that no two different requirements, `r1` and `r2`, have the same `entity` if one of them has subtype `'ENUM'` and the other has type `'EXTENDS'`. The needed data about each requirement is accessed through class attributes which reference the requirement `entity` and boilerplate (`bplate`). In turn, the boilerplate has the information about its `type` and `subtype`. Notice how each class inherits from `Monitored`, so that their objects can be associated, through `EventHook`, with handlers from trees having them in their scopes.

4.3 EVALUATION

In this section some evaluation of the implemented library is presented. The evaluation objectives were to 1) confirm that certain events triggered the expected tree transformations and reevaluations (or simply, transitions), to 2) demonstrate and compare the performance of different transitions for a given consistency rule, and to 3) evaluate, for a given transition, the impact in performance that the complexity of the rule may have. After this analysis, all results are discussed, in particular the cost of lazy evaluation, that is, of maintaining minimum scopes.

Every test was run several times in order to get more accurate and stable results. Some empiric correctness analysis was also done by always comparing the logical values of each tree with the ones from equivalent consistency rules defined and checked using Python assertions.

4.3.1 *Transition definition*

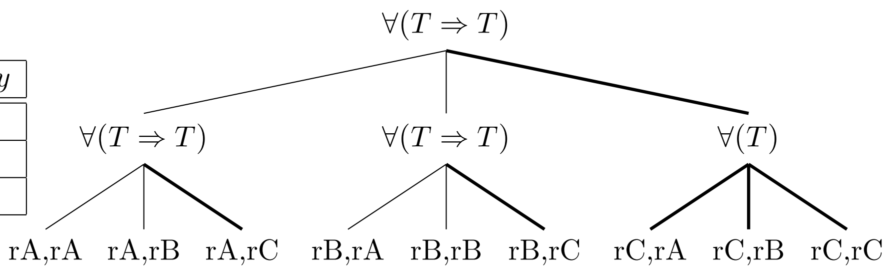
When a change is made to a model, being an addition, deletion or update, that change is associated with a model element/property. As we saw previously, an event occurring in one element is associated with handlers from leaf nodes belonging to trees which have that element included in their scope. Once triggered, an handler perform a tree transition which depends on the current state of the tree. These transitions correspond to tree transformations and reevaluations. Thus, a transition can be seen as a function from a pair of tree state and event, to another tree state resulting from a deterministic transformation and reevaluation:

$$transition: \;\; (tree, event) \Rightarrow tree'$$

Next, different transitions are tested for consistency trees relative to the rule presented in Subsection 4.2.4. Alongside an explanation for each state transformation, a complexity analysis is done and its result corroborated by testing the transition performance for increasing scope sizes. Yet, lets first understand how each tree is going to be represented.

The consistency tree depicted in Figure 10 is a simplified version of the actually generated tree for a model with three consistent requirements. This simplification is done for a clearer visualisation and interpretation of each transition. Only the nodes corresponding to the universal quantifiers are shown. In fact, all tested requirements satisfy the left operands (lines 2 and 5) of the first two implications (`'ENUM'` is actually a subtype of `'EXTENDS'`), so these can be omitted. Furthermore, the quantifiers ultimately determine the volume of the tree. The subtrees corresponding to the third implication here are abstracted as leafs by only depicting the two requirements being compared, the left one coming from the range of the outer quantifier, the right one coming from the range of the inner quantifier.

Requirement	*Entity*
rA	X
rB	Y
rC	**Z**



$\forall(T \Rightarrow T)$
$\forall(T \Rightarrow T)$
$\forall(T \Rightarrow T)$
$\forall(T)$
rA,rA
rA,rB
rA,rC
rB,rA
rB,rB
rB,rC
rC,rA
rC,rB
rC,rC

Figure 10: Adding consistent requirement rC

4.3.2 *Transition 1 - tree increment*

The first tested transition is shown in Figure 10, where requirement `rC` is added to a model of two consistent requirements, `rA` and `rB`. The thin line corresponds to the starting state of the tree, while the thick line represents new branches in the resulting state. As we can see, a subtree comparing `rC` with all requirements is appended to the right of the outer quantifier. Besides, the other subtrees, corresponding to `rA` and `rB`, gained an extra branch for comparison with `rC`, as `rC` is also an element included in the range of the inner quantifier. This particular situation where ranges overlap can originate redundant evaluations, so potential optimisations may be subject for future work.

Whenever a new branch is added to a consistent universal quantifier (or inconsistent existential quantifier, for direct comparison with the pseudo code from Section 4.2) the latter must be reevaluated. This reevaluation is performed by 1) validating the quantifier, which in turn results in validating newly created branches, 2) updating the quantifier's truth value accordingly, performing tree transformations if necessary (for lazy evaluation purposes), and finally by 3) propagating the reevaluation to the parent node if the quantifier's logical value has changed. The incrementality advantages are leveraged in the first step, where already existing branches do not need to be evaluated, as their state is reused instead. Reevaluations of nodes are represented by an arrow between the starting and resulting truth values. A single truth value with no arrow appears in newly created nodes, evaluated for the first time. The resulting model is still consistent since all the entities remain different from each other.

A high-level complexity analysis of each tested transition was done, corroborated by a performance test for increasing scope sizes. The complexity of an algorithm corresponds to the number of operations performed as a function of input size. Consider the input size n to be the number of requirements in the initial state, and one operation to be the processing of a leaf node (in Figure 10 corresponding to a subtree comparing two requirements), either its addition, deletion, validation, or any combination of these. Thus, for the transition seen we have

$$(n+1) + n = 2n + 1 = O(n)$$

, where $(n+1)$ and n come, respectively, from processing the right subtree (relative to `rC`), and processing every extra branch added to the other already existing subtrees. This equates to a linear complexity $O(n)$, resulting in a time execution directly proportional to the input size, fact corroborated by the performance results presented in Figure 11.

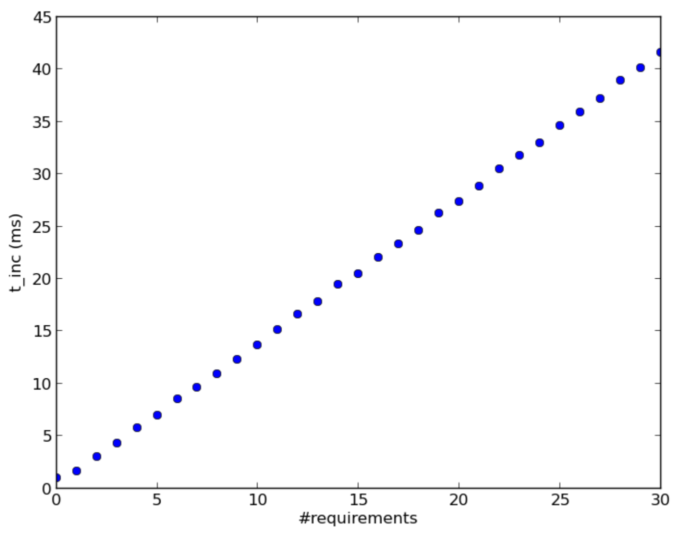


Figure 11: Requirement addition time for increasing scope

4.3.3 *Transition 2 - collapsing of the tree*

The second transition tested was to add an inconsistent requirement `rC` to the same starting consistent tree of two requirements. In Figure 12 we can see that, this time, the right subtree, relative to `rC`, has its two rightmost branches marked as discarded (dashed line). This is due to the fact that, since an inconsistent pair violating the universal quantifier was found (entities from `rC` and `rA` are equal), according to the lazy evaluation logic, every other branch of the quantification must be ignored. This consequently also happens for the outer quantifier as its truth value turns false. Therefore, both the other subtrees are discarded as well. All these deactivations make the tree collapse into a single branch witnessing the inconsistency cause.

Requirement	*Entity*
rA	X
rB	Y
rC	**X**

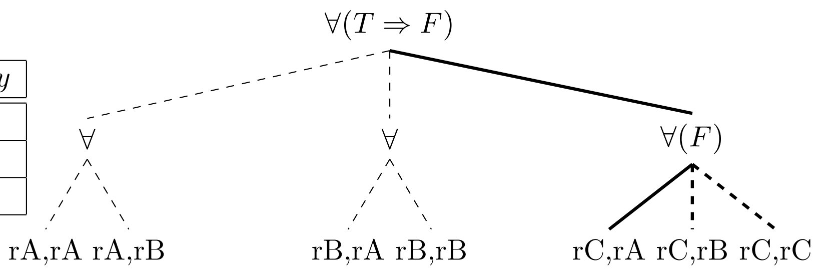


Figure 12: Adding inconsistent requirement rC

For this second transition we have

$$(n+1) + n^2 = O(n^2)$$

, where $(n+1)$ and n^2 come, respectively, from processing the rightmost subtree, and processing every other branch. Notice that deactivation is also a type of operation which has to reach each leaf in order to unbind their handlers. Also realise that, unlike what happened for the first transition, here no extra branch is added to already existing subtrees. This happens since the outer quantifier deactivates the inner ones before their handlers notice the event. The previous expression equates to a quadratic complexity $O(n^2)$, resulting in a time execution proportional to the square of the input size, as corroborated by the performance results presented in Figure 13. It should be noted that this cost can be amortised by other more efficient transitions.

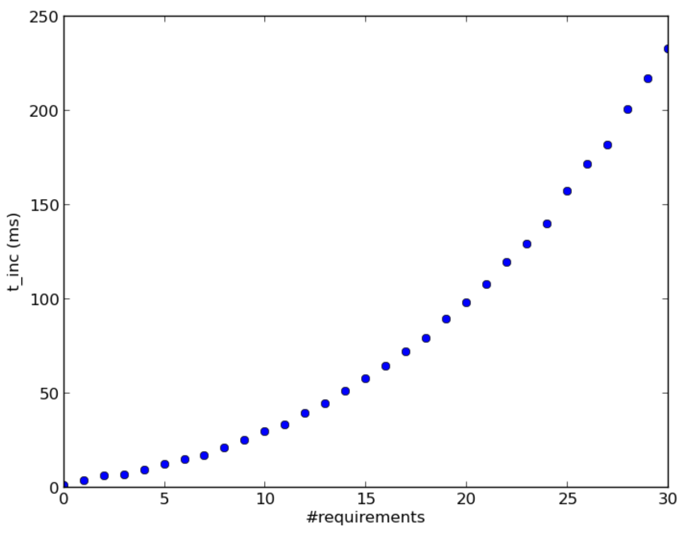


Figure 13: Collapse time for increasing scope

Requirement	*Entity*
rA	X
rB	Y
rC	X
rD	**Z**

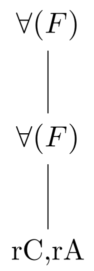


Figure 14: Adding out of scope requirement rD

4.3.4 *Transition 3 - out of scope event*

The third transition tested was to take the resulting state of the second transition and perform an event outside the scope of the consistency tree. As depicted in Figure 14, this is done by adding requirement `rD`. As the scope of the starting tree is composed only of requirements `rC` and `rA` (since their state alone is sufficient to determine the overall consistency), any event that does not concern these two requirements will fall out of scope, thus resulting in a void transition.

This time one has

$$O(1)$$

, which equates to a constant complexity, resulting in the same time execution regardless of the input size. This is corroborated by the performance results presented in Figure 15. This figure does not reveal an exactly horizontal line, because the time values are almost instantaneous and the axis scale used quite accurate.

4.3.5 *Transition 4 - rebuilding the tree*

The fourth transition tested consisted of deleting the inconsistent requirement `rC`, added in the second transition. Figure 16 shows that this deletion triggers a reconstruction of the tree that recovers those branches which had been previously ruled out. This happens since the deletion takes place when the universal quantifier is in a false state, therefore becoming necessary to check what happened to other elements, as there may be other evidence of falsehood. Because all the remaining requirements (`rA` and `rB`) have different entities, a consistent state is reached.

For this transition we have

$$1 + (n-1)^2 = O(n^2)$$

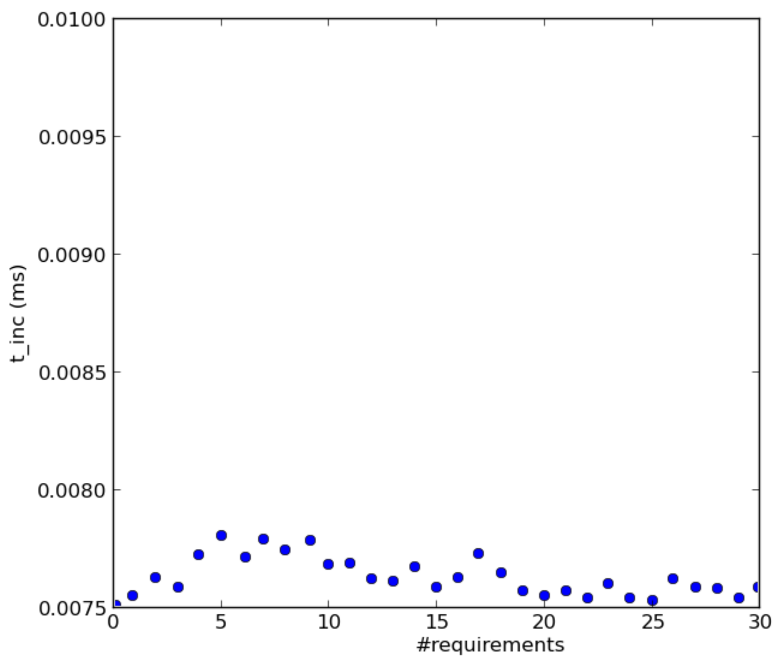


Figure 15: Out of scope event time for increasing scope

, where 1 and $(n-1)^2$ come, respectively, from deleting the single existing branch, and rebuilding/checking all the other combinations. This equates to a quadratic complexity $O(n^2)$, resulting in a time execution proportional to the square of the input size, as corroborated by the performance results presented in Figure 17.

4.3.6 *Formula complexity*

In order to demonstrate the impact that the complexity of a rule may have in performance, the same transition type was tested for trees/rules of increasing complexity, yet for the same

Requirement	*Entity*
rA	X
rB	Y
~~rC~~	~~X~~

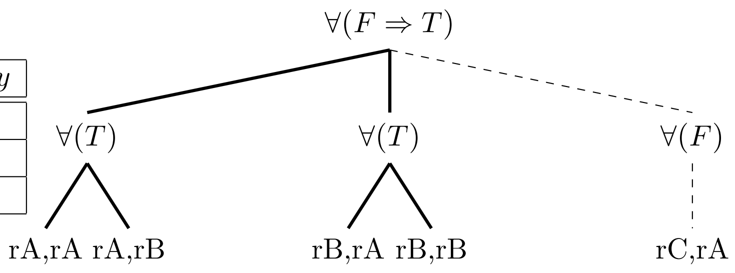


Figure 16: Deleting inconsistent requirement rC

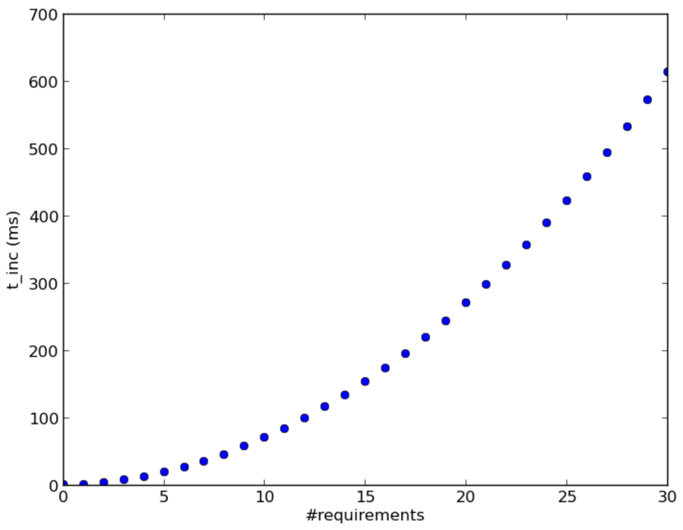


Figure 17: Rebuilding time for increasing scope

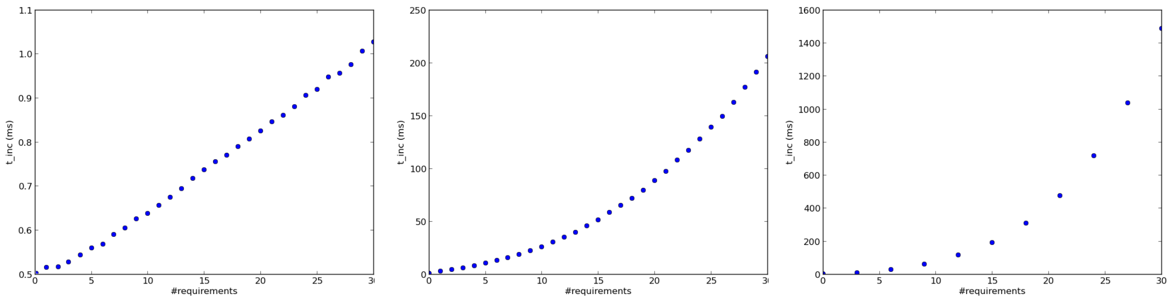


Figure 18: Transition times for (left to right) one, two, and five nested quantifiers

set of scope sizes. The transition type chosen was the collapsing of a tree, as happened in the second example tested. This is done by adding an element which makes all other branches unnecessary. The metric used as a measure of the complexity of a rule was the number of quantifiers. Figure 18 shows, from left to right, the times obtained for one, two and five quantifiers. As expected, it becomes clear that the more complex a rule is, the less scalable to verify it becomes. For instance, the times for checking the more complex rule hit the order of seconds just for a relatively small scope of approximately twenty-five requirements.

4.3.7 *Discussion of results*

Testing different transitions, for the same consistency rule, has shown different performance behaviours between them. As the scope grows, that is, as it becomes necessary to evaluate more elements in order to check the consistency of the model, some have proved to scale better than others. Their different complexities resulted in a significant and increasing cost variability as the scope gets larger. Importantly, the particular transitions tested demonstrated that this variability can mainly be due to lazy evaluation. This mechanism is attained by keeping scope sizes to a minimum. As already pointed out in Section 4.1, as soon as any minimal set of states sufficient to determine the logical value of a rule is found, no other state will need evaluation until that set changes and is no longer sufficient for checking consistency. Therefore, any event relative to any state outside this minimum set must not trigger any transition in the tree. Only this way evaluations are made only when their results are needed, as a lazy approach advocates. This restriction is attained by discarding branches, which also makes the approach more efficient in terms of memory usage. Nevertheless, implementing this concept induces the observed variability. As shown, while an event which falls out of scope does not trigger any transformation or evaluation, one that requires the tree to collapse or be rebuilt can be quite expensive to check.

Maintaining minimum scopes is somewhat arguable. An incremental approach by definition reuses previously computed states to avoid bulk evaluations. Discarding branches, although leveraging laziness advantages, it dumps previously computed states and ignores future updates, fact which sometimes will make bulk evaluations necessary for recovering the states of all the ignored parts of the model. Moreover, as evidenced, large scopes and complex rules can make this maintenance very expensive. In this sense, lazy evaluation and incrementality somehow conflict. In the other hand, if an eager evaluation is followed instead, where all elements are observed which have the potential to be included in the scope, there is no need to rebuild and evaluate branches when the states of those elements eventually become relevant (collapses are also avoided). In this setting, it would be expected to observe a lower variability in checking times. However, this approach implies that the number of evaluations performed are maximum, as every potentially relevant event must be tracked, case in which a great part of them may end up being useless, since the concerned elements may never become pertinent, or only very rarely. Another implication of this eagerness is a greater memory usage, since trees would be always at their maximum size.

In conclusion, as there are many variables at stake, such as the complexity of the formulae, the size of the scopes (relative to the model as a whole), and available memory, it seems reasonable to test both approaches in real contexts to determine which shows better performance in each particular case. A probable trade-off seems to be a better overall performance with an eager approach, for those cases where few model elements and considerably complex rules (with several nested quantifiers) prevail, while laziness would better fit to situations where a large number of model elements are constrained by simpler constraints. A considerable number of case studies from PROVA may help decide which particular technique should be used in the platform.

5

GENERATING REPAIR TREES

Repairing inconsistencies can be much harder than detecting them because the number of repair alternatives grows exponentially with the complexity of the design rule and the number of model elements involved (Reder and Egyed, 2012a). In this chapter the repair method introduced in Reder and Egyed (2012a) is presented as an extension to the previous checking mechanism that allows, for a given consistency tree, the generation of a corresponding repair tree. Section 5.1 presents an overview of this repair technique. In Section 5.2, the library of Section 4.2 is extended with a repair method which propagates through the tree in a top-down manner, generating alternative sequences of modifications, also arranged as a tree. In Section 5.3, this new functionality is tested for a slightly modified version of the consistency rule presented in Subsection 4.2.4, results are discussed, and a strategy for interpreting and simplifying repair trees is suggested. Finally in Section 5.4, other challenges to this repair approach are also discussed, as overlaps, negative side-effects and automation.

5.1 GENERATION MECHANISM

In Reder and Egyed (2012a) a technique for repairing models is introduced which turns out to be an extension to the method of incremental verification studied in Chapter 4. This technique generates a repair tree from an inconsistent check tree, the former exhibiting all the minimal sequences of abstract modifications to the model which make the latter shift to a consistent state. This section presents an overview of the generation mechanism.

5.1.1 *Expected logical value*

Generating a repair tree requires a repair function $R(t, E)$ which receives a consistency tree t and a Boolean result E representing the expected logical value for its root. This repair function requires that the consistency tree is already evaluated, since its operation depends on the comparison of the expected result E with the actual observed logical value, let us say, O. The procedure starts by checking if E differs from O, in which case the tree is inconsistent

a	b	$O(a \wedge b)$	E	$R(a \wedge b, E)$
f	t	f	t	$R(a, t)$
t	f	f	t	$R(b, t)$
f	f	f	t	$R(a, t)$ *and* $R(b, t)$
t	t	t	f	$R(a, f)$ *or* $R(b, f)$

Table 2: Repairing $a \wedge b$

and must be repaired. The repair tree is then generated based on the type of t, i.e. the logical operator of its root. This mechanism propagates recursively in a top down fashion, stopping by repairing leaf nodes or when E equals O. The leaves of the repair tree correspond to actual repair suggestions for the respective leaves in the check tree. Thus, each repair either suggests modifying a property of an element, adding an element to the range of some quantifier, or removing it instead.

For each recursive call, computing E is simple. The expected result for the top node of t is always true, as the design rule is expected to evaluate to true to be consistent. The value of E propagated down to the child nodes depends on the parent's type. For instance, if a conjunction is expected to be true (E = true) then its subtrees are expected to be true also, while in the case of negation, the opposite value $\neg E$ is expected for its single child.

5.1.2 *Eliminating false and non-minimal repairs*

Computing the repair alternatives for any given logical operation is more complex. For instance, Table 2 enumerates all possible repair actions for a conjunction, based on the values of E and O. If O is false and E is true, then there are three possible repair alternatives: 1) repair the first operand, $R(a, t)$, 2) repair the second operand, $R(b, t)$, or 3) repair both, $R(a, t)$ *and* $R(b, t)$. This list appears reasonable, however, it may contain incorrect repair alternatives and non-minimal repair actions. For example, if a is true already, then 1 is incorrect because it would repair something that is not broke, and 3 is non-minimal as it would repair more than necessary. Therefore, to identify the appropriate choices, repair alternatives are guarded. For example, if it is known that a is true, b is false and the expected result E is true, then 2 is the right choice. If E is false and O is true, then the repair alternatives change because either a or b need to be false, in which case $R(a, f)$ *or* $R(b, f)$ would be the proper repair. By eliminating false and non-minimal repairs, the approach vastly reduces the number of repair alternatives, however, not necessarily the exponential growth. As the extension to the library is described in Section 5.2, the repair method behaviour for every other supported logical operator is also explained.

Every node in a repair tree aside from the leafs ends up being an *and* or an *or* repair node, respectively representing the conjunction (sequence) and disjunction (alternative) of their

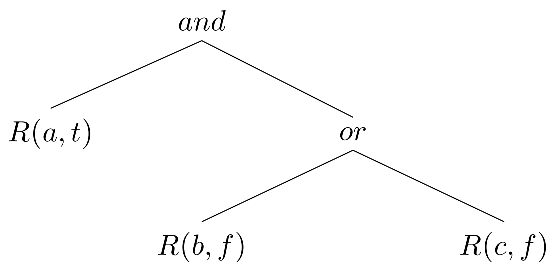


Figure 19: $R(a \wedge \neg(b \wedge c), t)$, where a is false, and b and c are true

repair subtrees. Since it was generated from a check tree t, each repair tree also reflects the syntactic structure (at least partially) of the concerned design rule (see Figure 19). According to the authors, this makes the repair tree intuitive to understand. Apparently, this is not always the case, as discussed further ahead in Section 5.3.

5.2 EXTENDING THE LIBRARY

The implemented check library presented in Section 4.2 was extended to support the generation of repair trees, by following the method described in the previous section.

As previously explained, generating a repair tree requires a repair function which receives some consistency tree and its expected logical value. Since this procedure behaves differently depending on the type of the root node, it becomes suitable to take advantage of polymorphism, by having separate implementations for each node type, yet with the same method signature. To this end, a repair method with a specific implementation was added to each class, the expected logical value being the only necessary argument. The invocation of the repair method is propagated in a top-down fashion, according to a specific control logic which eliminates false and non-minimal repairs, as already seen for the particular case of conjunction (Table 2). In this section, the implementation of this logic is presented for each supported operator.

5.2.1 *Repairing each logical operator*

Any implementation of the method `repair` must begin with the comparison of the observed logical value (from now on referred to as the tree's `result`) with the expectation (argument `e`). When these values match, the repair propagation stops.

For `Negation` the logic is simple. Here, the repair is immediately propagated to the sub-formula with the expected value negated (line 4):

```
1 class Negation(Formula):
  def repair(e):
    return None if result == e
    return childFormula.repair(not e)
```

As seen, for `Conjunction` things get more complex, since false and non-minimal repairs must be avoided:

```
 1 class Conjunction(Formula):
 2   def repair(e):
 3     return None if result == e
 4
 5     if e == false:
 6      return (leftChildFormula.repair(e) OR rightChildFormula.repair(e))
 7
 8     if e == true:
 9       if leftChildFormula.result == false:
10         r1 <- leftChildFormula.repair(e)
11         activate rightChildFormula with no triggers
12         evaluate rightChildFormula
13         if rightChildFormula.result == false:
14           r2 <- rightChildFormula.repair(e)
15           repair <- (r1 AND r2)
16         else:
17           repair <- r1
18         deactivate rightChildFormula
19       else:
20         ...
21     return repair
```

Lines 5 and 6 correspond to the last line of Table 2, where `e` is false and the observed `result` is true. Here the proper suggestion is the alternative between repairing (to false) the left subtree and the right subtree, so an `OR` node is returned.

Lines 8 to 20 address the other three possible repair cases. If `result` is false, one of the branches is also false and the other is deactivated thanks to the lazy evaluation mechanism. Only the algorithm for an active left branch is shown (lines 10 to 18), the control logic for the reverse case being analogous. Since the branch is false and `e` is true, at least this branch must be repaired. Note, however, that there is the need to reactivate and evaluate (lines 11 and 12) the other branch (potentially expensive) in order to determine if its `result` is also false, in which case it would be necessary to repair it as well (lines 14 and 15). Since it is crucial not to affect the independent checking mechanism, this activation must be only temporary, not enabling any triggers (line 11 and 18).

For `Equals`, ignoring the existence of constants, the repair simply corresponds to the alternative between repairing the first access and repairing the second access, thus represented as a repair tree of type disjunction (`OR node`). Notice that repairing an instance of `Access` does not requires an expected value, since it is not a subclass of `Formula`, but rather a terminal node. To make things clearer another method name is used, namely `fix`. This method, next explained in Subsection 5.2.2, generates three types of repairs depending on its caller and on an optional argument:

```
1 class Equals(Formula):
2   def repair(e):
3     return None if result == e
4     return (leftAccess.fix() OR rightAccess.fix())
```

For `Exists`, if `result` is false, to make it true (lines 5 to 8) it will be necessary to make at least one of its branches also true, or, alternatively, to add a new branch satisfying the quantification.

```
 1 class Exists(Formula):
 2   def repair(e):
 3     return None if result == e
 4
 5     if e == true:
 6       for every branch:
 7         repair <- (repair OR branch.repair(e))
 8       return (repair OR source.fix('add'))
 9
10     else:
11       buildFormula in temporary mode
12       for every branch:
13         evaluate branch
14         if branch is true:
15           alt <- (branch.repair(e) OR
16                   source.fix('del', branch))
17           repair <- (repair AND alt)
18       return repair
```

In the opposite case (lines 10 to 18), ie, for an expected result of false but where at least one branch is true, it will be necessary, for every consistent branch, to repair it or delete it. In this case, all branches (other than the one satisfying the quantifier) must be temporarily reactivated and evaluated (again, without interfering with the checking mechanism), so that all other cases that also satisfy the quantifier are found. As discussed in Section 4.3, this rebuild operation can be quite expensive.

5.2.2 *Fixing terminal nodes*

Next, the repairs of the leaf nodes are presented. For these cases a different method name is used since it does not require an expected value, representing an end point of the recursive chain. It is here where the three possible actual repair suggestions will be generated, namely `MODIFY`, `ADD` and `DELETE`. As expected, `MODIFY` is generated in an instance of `Access`, while `ADD` and `DELETE` repair the `Source` of quantifiers.

Since an `Access` represents a chain of accesses to properties, any modification to that chain will eventually change the final value accessed. Thus, the desired repair will be the disjunction of all possible modifications to the chain. To better understand this, consider for example that one wants to change the grandfather of a person, accessed by `person.father.father`. This can be achieved by directly changing that person's grandfather, that is, the `father` property of `person.father`, but also by changing the father of that person, that is, the `father` property of `person`. Therefore, each modification refers to a pair of element reference and respective attribute (line 4):

```
1 class Access():
2   def fix():
3     for every ref, att in access_list:
4       repair <- (repair OR ('MODIFY', ref, att))
5     return repair
```

In case of `Source`, `fix` is invoked with an argument indicating the need to add or remove an element:

```
1 class Source():
2   def fix(operation, ref):
3     if operation is 'add':
4       return ('ADD', parent.formulaString)
5     else:
6       return ('DELETE', ref)
```

If adding an element is necessary, the `ADD` message and the constraint that the new element should satisfy are returned (line 4). It will be up to the user to somehow add this element to the set. In the case of deletion, `DEL` is returned along the element reference (line 6). Since this repair is deterministic, it can be executed automatically.

Requirement	*Entity*
rA	X
rB	X
rC	**Y**

Table 3: Inconsistent model

5.3 REPAIR EXAMPLE

In this section a repair tree is generated from an inconsistent tree of a slightly modified version of the rule presented in Subsection 4.2.4. The generated structure is explained and the way to interpret it is discussed. Following this, a strategy is presented for simplifying the repair tree by eliminating redundant information and uncovering those suggestions with greater repair potential, this way making it more intuitive to the user.

5.3.1 *Tree structure*

Consider that one intends to ensure that any requirement `r1` with subtype `ENUM` must have the same entity of any requirement `r2` of type `EXTENDS`:

```
forall r1 in requirements : (
        r1.bplate.subtype = 'ENUM'
        ->
        forall r2 in requirements : (
                r2.bplate.type = 'EXTENDS'
                ->
                r1.entity = r2.entity
        )
)
```

Consider also that one adds a requirement `rC` with entity `Y` to a model with two requirements, `rA` and `rB`, with the same entity `X` (see Table 3). Assume that these three requirements have the subtype `ENUM`, and that `EXTENDS` is the supertype of `ENUM` itself. This operation results in a model with three requirements which violates the consistency rule, since there are now different entities (`X` and `Y`) when, in the scenario considered, they should all be the same.

A repair tree was generated by invoking the `repair` method on the violated consistency tree. To facilitate the comprehension of the tree structure as a mirror of the rule's syntax, each repair node was labelled with the check node from which it was generated - `OutForall` for the outer quantifier, `InForall` for the inner quantifier, `->` for any implication, `A` for any `Access`, `=` for any `Equals`, `InSource` for the source of the inner quantifier, and `OutSource` for the source of the outer quantifier. When appropriate, a label is followed by the requirement the repair node respects to (as in line 2):

```
 1 AND - OutForall
 2    OR - OutForall (rA)
 3    .  OR - ->
 4    .  .  OR - A
 5    .  .  .  ('MODIFY', rA, 'bplate') - A
 6    .  .  .  ('MODIFY', bplate, 'subtype') - A
 7    .  .  OR - InForall (rC)
 8    .  .     OR - ->
 9    .  .     .  OR - A
10    .  .     .  .  ('MODIFY', rC, 'bplate') - A
11    .  .     .  .  ('MODIFY', bplate, 'type') - A
12    .  .     .  OR - =
13    .  .     .     ('MODIFY', rA, 'entity') - A
14    .  .     .     ('MODIFY', rC, 'entity') - A
15    .  .     ('DELETE', rC) - InSource
16    .  ('DELETE', rA) - OutSource
17    OR - OutForall (rB)
18    ...
19    OR - OutForall (rC)
20    ...
```

The generated repair tree is a conjunction (`AND` node) of three subtrees (lines 2, 17 and 19), each one concerning one of the requirements. In turn, these subtrees are disjunctions (`OR` node), here labelled as `OutForall (rA), OutForall (rB), and OutForall (rC)`. In order to understand why this is the case, consider the `else` block of the `repair` method of class `Exists`. Since an universal quantifier is obtained from an existential quantifier through rule $\forall x P(x) \equiv \neg\exists x \neg P(x)$, an expected value of true for the former equates to an expected value of false for the latter. An expected value of false will then make the `else` block run. As already explained in Section 5.2, this block generates a repair tree which is a conjunction of disjunctions, one disjunction per branch which evaluates to true. Since a branch is $\neg P(x)$ instead of $P(x)$, we will end up with one disjunction for every x requirement for which $P(x)$ evaluates to false, P being the outer implication. In this case all requirements satisfy the latter condition (as we shall see next) so three `OR` subtrees are generated.

Let us now focus on the subtree for requirement `rA`, with root `OR` $-$ `OutForall (rA)`, line 2. This repair tree presents two repair alternatives, one for repairing the requirement (line 3) and another one for deleting it (line 16). The latter is already a specific repair suggestion (a leaf node), while the former is another subtree with two possibilities. Repairing the requirement can be done by either modifying its subtype (lines 4 to 6), in which case it would violate the first operand of the outer implication (which in turn would become true anyway), or pushing the repair further to the inner quantifier (line 7). The tree of the inner quantifier was also generated in the `else` block, but since only requirement `rC` is inconsistent with `rA`, this subtree is not a conjunction of disjunctions, being rather one single disjunction. Analogously, here one can delete `rC` (line 15) or push the repair even further by either making `rA` violate the first operand of the inner implication (lines 9 to 11), or repairing the violated equality (lines 12 to 14), the latter being an alternative between modifying the entity from `rA` or the entity from `rC`.

The repair subtree for requirement `rB` is analogous. In order to simplify the output, hereafter terminal subtrees are represented with labels. `TrX` is used to represent the modification of the type or subtype of any requirement `rX`. `MrX` is used to represent the modification of any other attribute, while `DrX` stands for requirement deletion. Notice that `rC` is the only requirement inconsistent with `rB`, therefore being the one which appears in this subtree, as also occurred for `rA`:

```
...
OR - OutForall (rB)
   OR - ->
   .  TrB - A
   .  OR - InForall (rC)
   .     OR - ->
   .     .  TrC - A
   .     .  OR - =
   .     .     MrB - A
   .     .     MrC - A
   .     DrC - InSource
   DrB - OutSource
...
```

The repair subtree for `rC` is a bit more complex because this requirement is inconsistent with the other two. Following the same pattern, the difference here is that the inner quantifier (lines 5 to 19) is a conjunction of two disjunctions, one for `rA` and another one for `rB`:

```
1   ...
2   OR - OutForall (rC)
3      OR - ->
4      .  TrC - A
5      .  AND - InForall
6      .      OR - InForall (rA)
7      .      .  OR - ->
8      .      .  .  TrA - A
9      .      .  .  OR - =
10     .      .  .     MrC - A
```

```
    .       .  .     MrA - A
    .       .  DrA - InSource
    .       OR - InForall (rB)
    .          OR - ->
    .          .  TrB - A
    .          .  OR - =
    .          .     MrC - A
    .          .     MrB - A
    .          DrB - InSource
    DrC - OutSource
```

5.3.2 *Interpreting and simplifying repair trees*

Trying to analyse the entire tree was an arduous and unintuitive task, even for a small scope and relatively simple rule. Although all those nested nodes mirror the syntax structure of the rule, quickly realising what are the actual repair combinations seems to be much more important in what respects to intuitiveness. As shown in figure 20, the first simplification suggested is to flatten every subtree having only `Or` nodes into a single `Or` root with all alternatives listed. However, despite making more obvious to the user what are the possible repair combinations, the resulting repair tree still has some problems, concerning not only intuitiveness. Let us investigate the matter further.

First it is important to understand how to read a repair tree. We could consider all possible combinations a repair tree allows, but a major part of these combinations are redundant or even senseless. The repair tree in figure 20 has a total of 648 combinations, resulting from multiplying the number of alternatives of every disjunction at the same tree level. However there is a high level of redundancy due to hidden positive side effects. A positive side effect occurs when a certain change to the model repairs more than one subtree (or even entire repair trees). For instance, considering repair `MrC`, that repair alone has the potential of repairing the whole tree. Yet this high repair potential is somehow hidden throughout different branches when the user's best interest is to immediately spot these cases.

Due to the positive side effects, a large number of combinations become undesirable. For instance, choosing `TrA` in the topmost disjunction repairs that `OR` node but also repairs the other disjunction where it reappears lower in the three. Thus any given alternative for repairing this other disjunction become redundant or, even worst, potentially unnecessary. Considering this problem, the way a repair tree should be read is by immediately considering the positive side effects of a chosen suggestion, this in order to discard all further options belonging to consequently repaired trees. Yet, another existing problem has to do with non minimal repairs, even if one correctly reads a repair tree by ignoring certain combinations. Consider, for instance, choosing `TrA` in the topmost disjunction and `MrC` in the following `OR` node. As

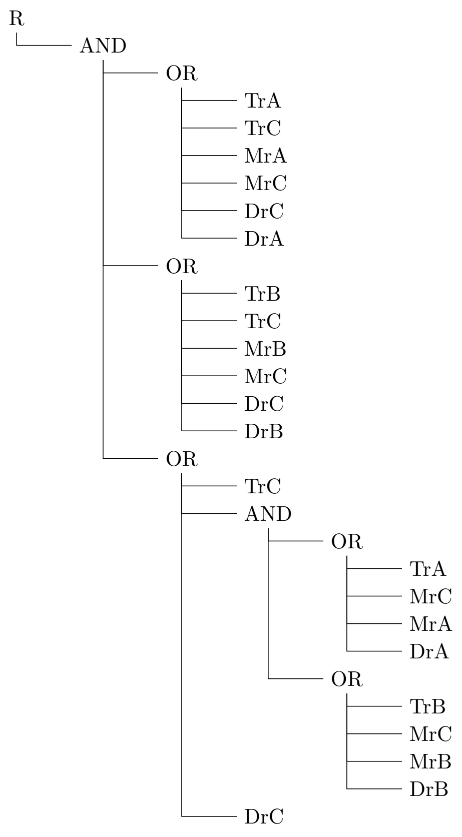


Figure 20: Flattened repair tree

previously seen, `MrC` has the potential for repairing the entire tree, making `TrA` unnecessary. Unless one considers that `TrA` is somehow cancelled or that there is no specific choosing order (case in which considering `MrC` first would discard `TrA` from the user's choice), the resulting repair becomes non minimal. In short, not only the repair tree has a lot of redundant and cluttered information, even worse, all this analysis is left up to the user.

An alternative is to try to simplify the generated repair tree using simple rules from propositional logic. For instance, using the distributive property one can merge positive side effects in higher level nodes, this way reducing redundancy and making more obvious to the user their greater repair potential. Using this and other simple laws, the generated tree was reduced to a much more simple and intuitive structure of only 12 combinations to reason about (see figure 21):

$$
\begin{gathered}
(T_{rA} \vee T_{rC} \vee M_{rA} \vee M_{rC} \vee D_{rC} \vee D_{rA}) \ \wedge \ (T_{rB} \vee T_{rC} \vee M_{rB} \vee M_{rC} \vee D_{rC} \vee D_{rB}) \\
\wedge \ (T_{rC} \ \vee \ ((T_{rA} \vee M_{rC} \vee M_{rA} \vee D_{rA}) \wedge (T_{rB} \vee M_{rC} \vee M_{rB} \vee D_{rB})) \ \vee \ D_{rC}) \\
\Longleftrightarrow \text{ distributive } (M_{rC}) \\
(T_{rA} \vee T_{rC} \vee M_{rA} \vee M_{rC} \vee D_{rC} \vee D_{rA}) \ \wedge \ (T_{rB} \vee T_{rC} \vee M_{rB} \vee M_{rC} \vee D_{rC} \vee D_{rB}) \\
\wedge \ (T_{rC} \ \vee \ (M_{rC} \ \vee \ ((T_{rA} \vee M_{rA} \vee D_{rA}) \wedge (T_{rB} \vee M_{rB} \vee D_{rB}))) \ \vee \ D_{rC}) \\
\Longleftrightarrow \text{ associative} \\
(T_{rA} \vee T_{rC} \vee M_{rA} \vee M_{rC} \vee D_{rC} \vee D_{rA}) \ \wedge \ (T_{rB} \vee T_{rC} \vee M_{rB} \vee M_{rC} \vee D_{rC} \vee D_{rB}) \\
\wedge \ (T_{rC} \ \vee \ M_{rC} \ \vee \ ((T_{rA} \vee M_{rA} \vee D_{rA}) \wedge (T_{rB} \vee M_{rB} \vee D_{rB})) \ \vee \ D_{rC}) \\
\Longleftrightarrow \text{ distributive + associative } (T_{rC} \ M_{rC} \ D_{rC}) \\
T_{rC} \ \vee \ M_{rC} \ \vee \ D_{rC} \ \vee \ ((T_{rA} \vee M_{rA} \vee D_{rA}) \ \wedge \ (T_{rB} \vee M_{rB} \vee D_{rB}) \\
\wedge \ ((T_{rA} \vee M_{rA} \vee D_{rA}) \ \wedge \ (T_{rB} \vee M_{rB} \vee D_{rB}))) \\
\Longleftrightarrow \text{ (associative + idempotent)} \\
T_{rC} \ \vee \ M_{rC} \ \vee \ D_{rC} \ \vee \ ((T_{rA} \vee M_{rA} \vee D_{rA}) \ \wedge \ (T_{rB} \vee M_{rB} \vee D_{rB}))
\end{gathered}
$$

5.4 CHALLENGES TO THE APPROACH

In this section, some other challenges to the presented repair approach are briefly described. Starting with overlaps between repairs, next we will see negative side effects, ending with a discussion about the extent to which automation is possible.

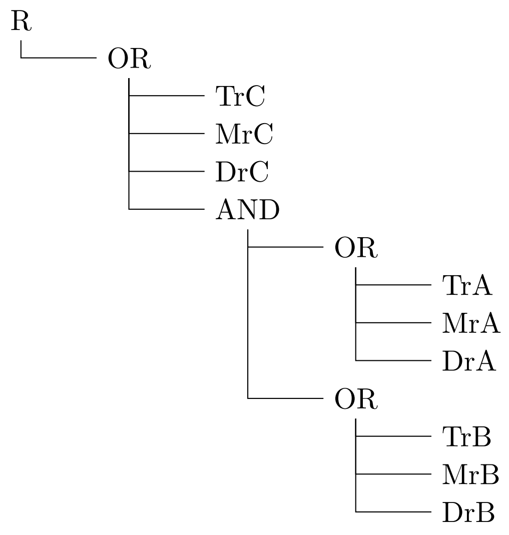


Figure 21: Simplified repair tree

5.4.1 *Overlaps*

A problem which can occur in certain repair combinations of a repair tree has to do with overlapping model changes. This happens when two repair actions have the model element and property in common or when the value of the property of one action is the model element of the other action. As an example of the latter case, deleting some requirement (`DrX`) would make it impossible to modify any attribute of that requirement (`MrX`). In the repair tree of the previous section this is not a problem, since the positive side effects of any deletion would discard further changes to the requirement in question. Yet, the problem may arise in other repair trees with different structures, or if an incorrect interpretation of the tree is done.

As a solution to this problem, Reder and Egyed (2012a) suggest to filter out these incorrect combinations. However, this solution can not simply reduce the tree presented to the user. Although not explicitly said, interactively restricting allowed repair combinations, as soon as the user chooses some action, seems to be the most reasonable approach.

5.4.2 *Negative side effects*

Repairs to inconsistencies may affect other design rules since they may solve other inconsistencies or they may cause other inconsistencies. As already mentioned, understanding these side effects is important during the repairing of inconsistencies to get an overview of the overall effect of a repair action on the model.

The previous section introduced positive side effects, in that one repair may also repair other inconsistencies (or their parts). Side effects may also be negative in that one repair may cause an inconsistency where there is none. Hence, it becomes necessary to identify all model elements affected by a choice in order to determine all consistency rule instances that are affected by that choice. Following this, Egyed et al. (2008) detect such potential side effects by checking whether a repair action of a repair tree references a model element in the scope of a consistent design rule. All repair operations in this condition are then eliminated resulting in refined repair plans. Yet, this solution seems to have some problems. In fact, there may be cases where a negative side effect should not be ignored. For instance, this can occur when repairing the model may actually involve successive repairs as the user builds upon his or her design (for example, more elements/relations become necessary as a consequence of the last change), or even remodels it differently (after all the user realises, thanks to the introduced inconsistency, that he/she wanted something else).

5.4.3 *Abstract repairs*

The generated fix operations presented are in fact abstract repairs, in the sense that it is up to the user to manually instantiate them. For instance, suggesting to add an element to a certain collection does not state whether the element should be created anew or should come from elsewhere. Suggesting to modify some property does not reveal what value one should assign to it. As they require human intervention these are not fully automatic repairs. Choice generation functions can however be used to generate possible values that specific fields of a model element may take (Egyed et al., 2008). Nevertheless, as considering all possible values would not scale, this approach typically only considers values that are already present in the model.

6 CONCLUSION

In this dissertation, a comparison of different existing model repair techniques was done, and a proposal was made for a taxonomy of relevant features for comparing these methodologies. A case study provided by Educed was implemented with the off-the-shelf model repair tool Echo, and an analysis was done assessing its suitability to be used as a model repair component in the PROVA platform. The incremental approach to model repair introduced in Reder and Egyed (2012a) was then implemented and evaluated as a promising alternative to Echo. Alongside some critical analysis, different aspects of the technique were evaluated, and a strategy for interpreting and simplifying repairs was suggested.

6.1 MAIN RESULTS

The following list summarises the main results obtained throughout this work:

- The proposed taxonomy for comparing model repair approaches includes features concerning incrementality, user interaction, automation, completeness, manual effort, inconsistencies resolution, side effects, *least-change* behaviour, control over repairs and underlying technique.
- Among all compared approaches, the one from Reder and Egyed (2012a) appears to be the one which scales better, since it is iterative, does not necessarily repair all inconsistencies at once, and follows a syntactic-based analysis. However, the approach is not fully automatic and no control over repairs generation is given. Although less scalable, Echo (Macedo et al., 2013) is fully automatic, generating repaired models which are as close as possible to the original, and provides control over repairs generation through the specification of allowed edit operations.
- Echo was quite intuitive to use and have proved to have a good support for OCL. Although useful, a *least-change* automation does not necessarily yields the desired repairs firstly. Besides, the performance times were not satisfactory enough for an approach with scalability issues such as Echo. The tool does not reuse results from previous

checks, and was unable to pinpoint which particular rules were violated. These problems led to the conclusion that, currently, Echo may not be well suited to be integrated into the PROVA platform as a model repair component.

- Regarding the incremental approach, as the scope of a tree grows, some transitions have proved to scale better than others. Their different complexities resulted in a significant and increasing cost variability as the scope gets larger. Importantly, the particular transitions tested demonstrated that this variability can mainly be due to lazy evaluation. A probable trade-off seems to be a better overall performance with an eager approach, for those cases where few model elements and considerably complex rules prevail, while laziness would better fit to situations where a large number of model elements are constrained by simpler rules.

- Trying to analyse a generated repair tree can be an arduous and unintuitive task, even for a small scope and relatively simple rule. Although its nested nodes mirror the syntax structure of the rule, quickly realising what are the actual repair combinations seems to be much more important in what respects to intuitiveness. Flattening a repair tree makes it more obvious what are the actual repair alternatives. However, there is still a high level of redundancy and a large number of undesirable combinations due to existing positive side effects. These are hidden throughout different branches, when the user's best interest is to immediately spot these cases with high repair potential. For these reason, the way a repair tree should be read is by immediately considering the positive side effects of a chosen suggestion, this in order to discard all further options belonging to consequently repaired trees. Simplifying the flattened repair tree using simple rules from propositional logic can greatly facilitate the its interpretation. For instance, using the distributive property one can merge positive side effects in higher level nodes, this way reducing redundancy and making more obvious to the user their greater repair potential.

6.2 FUTURE WORK

As future work, the following goals are proposed:

- Develop the collection of comparison features into a more mature, well-defined, and hierarchically organised taxonomy.

- Compare more existing model repair approaches, performing some performance study when possible.

- Help in developing Echo, by implementing other realistic case studies.

- Collect and implement more case studies from PROVA in order to assess which particular incremental repair technique, lazy or eager, better suits the platform needs.
- Investigate more ways to simplify repair trees and innovative solutions for tackling associated challenges, such as negative side effects and lack of automation.

BIBLIOGRAPHY

Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 511–520. IEEE, 2008.

Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th international conference on Software engineering*, pages 176–185. ACM, 2005.

Alexander Egyed. Instant consistency checking for the UML. In *Proceedings of the 28th international conference on Software engineering*, pages 381–390. ACM, 2006.

Alexander Egyed. Fixing inconsistencies in UML design models. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 292–301. IEEE, 2007.

Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 99–108. IEEE, 2008.

A Hegedus, Akos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. Quick fix generation for DSMLs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 17–24. IEEE, 2011.

Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski. Guided development with multiple domain-specific languages. In *Model Driven Engineering Languages and Systems*, pages 46–60. Springer, 2007.

Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements engineering*. Springer, 2011.

Daniel Jackson. Software abstractions-logic, language, and analysis, revised edition, 2012.

Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. Model repair and transformation with Echo. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 694–697. IEEE, 2013.

Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 455–464. IEEE, 2003.

OMG. *MOF 2.0 Query/View/Transformation Specification (QVT), Version 1.1*, January 2011. Available at http://www.omg.org/spec/QVT/1.1/.

Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, pages 1–21, 2013.

Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 220–229. IEEE, 2012a.

Alexander Reder and Alexander Egyed. Incremental consistency checking for complex design rules and larger model changes. In *Model Driven Engineering Languages and Systems*, pages 202–218. Springer, 2012b.

Marcos Aurélio Almeida Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards automated inconsistency handling in design models. In *Advanced Information Systems Engineering*, pages 348–362. Springer, 2010.

George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of software engineering and knowledge engineering*, 1:329–380, 2001.

Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Model Driven Engineering Languages and Systems*, pages 1–15. Springer, 2007.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in model-driven software engineering. In *Models in Software Engineering*, pages 35–47. Springer, 2009.

Ragnhild Van Der Straeten, Jorge Pinna Puissant, and Tom Mens. Assessing the kodkod model finder for resolving model inconsistencies. In *Modelling Foundations and Applications*, pages 69–84. Springer, 2011.

Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 315–324. ACM, 2009.