Pragmatic model verification

Carlos Alberto Gonzalez Perez

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Thèse de Doctorat

Carlos A. GONZÁLEZ

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

Discipline : Informatique et applications
Laboratoire : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 9 Octobre 2014

École doctorale : 503 (STIM)
Thèse n° : 2014 EMNA 0189

Pragmatic Model Verification

JURY

Rapporteurs :	**M. Antoine BEUGNARD**, Professeur, Telecom Bretagne **M. Martin GOGOLLA**, Professeur, Universität Bremen
Examinateur :	**M^me^ Esther GUERRA**, Maître de conférences, Universidad Autónoma de Madrid
Directeur de thèse :	**M. Jordi CABOT**, Maître de conférences, HDR, École des Mines de Nantes

Acknowledgments

The completion of this thesis would have not been possible without the help and support of many individuals. These few lines are just an expression of my gratitude to all of them.

I would like to express my most special thanks and gratitude to my thesis advisor, Jordi Cabot, for all the support given throughout these years. Every time I got stuck, his advice helped me to move further. Every time I made a mistake, he offered me valuable and constructive feedback. And more importantly, his door was always opened for me, every time I needed to discuss anything. Without his help, this thesis would have not been possible.

I am also grateful to Robert Clarisó and Fabian Büttner for their help and cooperation on the development of some of the ideas presented in this thesis.

My thanks also go to the members of the jury, for finding the time to review my work, and for the valuable feedback provided.

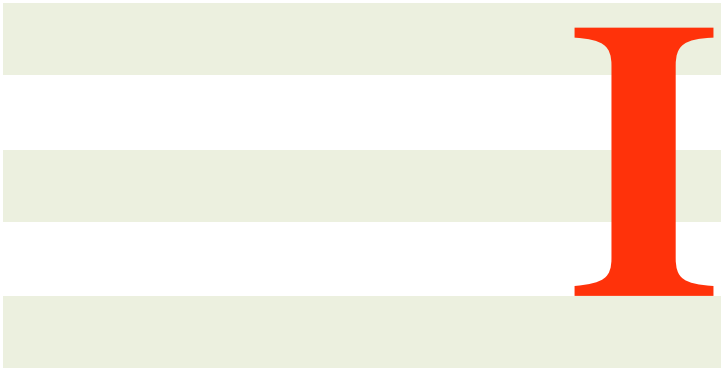
Thanks also to the members of the AtlanMod research team for creating such a loving working environment, and specially to Hugo Brunelière, for taking the time to review the French contents of this thesis.

Finally, thanks of course to my family. Their support is always there.

Contents

List of Tables

List of Figures



Introduction and Basic Concepts

1

Introduction

1.1 Motivation

Computers and software have become essential in the modern society, exerting an enormous impact in the way we produce and consume goods and services. There are examples everywhere: financial systems are highly computerized, the majority of money transactions being merely a data exchange among computers; transportation, from intelligent navigation systems to ships, trains, airplanes or cars that are fully loaded with electronic equipment; research, where activities like DNA sequencing or space exploration would not be possible without computers; healthcare, where computers store medical records, or are essential part of complex equipment such as monitoring systems or medical imaging systems; communication, with computers being an integral part of TV and radio broadcasting systems, electronic mail systems or video conferencing systems; recreation, where holiday packages or popular event tickets are mostly sold online; and so, the list goes on and on.

The immediate effect of all this is that, unsurprisingly, software industry has experimented an exponential growth in the last decades. What is more, far from slowing down, that tendency is accelerating. With modern societies constantly demanding for more and better products and services, which in its turn, requires more and more complex software systems, today, software industry is growing at an unprecedented pace. This growth implies that first, more people work in activities related to the software industry, and second, that new and more powerful tools and techniques appear every day to develop complex software as easy as possible.

Even though all this describes a bright present and a promising future for the software industry, truth be told, there are also some concerns, especially when it

comes to aspects like quality, absence of errors, reliability and the like. Although ensuring software correctness is not a new challenge [1], but an old one that software engineers continue to struggle with, the ever-increasing complexity of software projects, along with the enormous pressure to reduce costs and time to market are making it even more challenging. This is especially worrisome when looking at the consequences that software failures in critical systems can cause. A simple look at Wikipedia [1] reveals that, in the best-case scenario, these failures provoke substantial losses of money (as an example, the investment to solve the Y2K bug surpassed the amount of 100 billion dollars, in US only), but in the worst one, the cost is paid in the form of human lives.

Software industry is obviously well aware of these problems and, consequently, substantial research efforts have been and continue to be made to alleviate them. To date, the most popular trends to address this challenge are commonly referred to as software verification and software testing. Software verification comprises those approaches based on the use of formal analysis techniques to prove software correctness. Software testing, on the other hand, usually refers to those approaches that try to find errors in software by systematically running it with a set of inputs for which the expected output is known, and then comparing the actual outcome with the expected one. However, and despite the popularity of these approaches, research efforts have not only focused on the discovery of mechanisms to improve the reliability of software developed by traditional or conventional means. On the contrary, there are relatively new software development paradigms and methodologies that are also gaining traction, such as Software Product Lines (SPL) [2] or Model-Driven Engineering (MDE) [3], as promising ways of developing more reliable software.

In the particular case of MDE, the main idea is the utilization of models as first-class citizens of the software development process. In an MDE-based software development process, the software is not coded by hand, but by designing and creating a number of models to be successively and (semi)automatically transformed into more refined models, and eventually into the code comprising the new software system. This way, by increasing the abstraction level and reducing the effort made in labor-intensive and error-prone manual tasks such as coding, the amount of errors in the final software system may be reduced.

Although what MDE promotes sounds promising, truth is, an MDE-based software development process also requires the presence of additional mechanisms to try to ensure its reliability. When MDE is applied to the development of complex software, the complexity of models and model transformation involved in the process tends to increase. This turns their creation and edition into error-prone tasks that endanger the reliability of the whole process, and therefore the soundness of

1. http://en.wikipedia.org/wiki/List_of_software_bugs

the resulting software. Thus far, the research efforts made to alleviate this have consisted in trying to adapt software verification and software testing techniques to the reality of models and model transformations of MDE. Unfortunately, as of this writing, the tools and techniques resulting from these efforts do not enjoy a great deal of success. For this reason, in this thesis, we try to find out what are the reasons that are preventing a wider adoption of these tools, and propose new mechanisms and techniques to try to change this.

1.2 Objectives and Contributions of this Thesis

The objective of this thesis is to improve the landscape of approaches devoted to try to ensure quality and absence of errors in models and model transformations. In particular, the focus is on the analysis of approaches devoted to the verification of static models, which are arguably the models more commonly adopted at the time of describing the specification of a software system [4]. The intent is to develop new mechanisms and to make the existing ones more efficient, thus facilitating their wider adoption. Additionally, the role that these approaches can have at the time of testing model transformations is also studied. As a consequence of this, several techniques where verification approaches are used for the generation of test data are also proposed.

More specifically, the contributions of this thesis are:

- A mechanism aimed at ensuring static model correctness based on constraint programming called EMFtoCSP.
- Two mechanisms aimed at improving the efficiency of model verification approaches.
- A mechanism devoted to the generation of input test data for model transformation testing, based on the analysis of model transformation internals.
- A mechanism devoted to the generation of input test data for model transformation testing, based on the analysis of the model transformation specification.

1.3 Outline of the Thesis

The rest of this thesis is structured as follows:

- **Chapter 2** introduces some basic terminology that will be commonly employed throughout this thesis. It constitutes, along with this chapter, the introductory part.
- **Chapter 3** presents the current landscape in the field of static model verification. The existing tools and techniques are analyzed and classified, some

weaknesses are identified, and several improvement proposals are suggested.

- **Chapter 4** presents EMFtoCSP, a tool for the verification of static models based on Constraint Programming. The tool belongs in a family of model verification tools that limit the search space where to look for a solution of the verification problem. This family of tools, commonly referred to as bounded verification approaches, is gaining popularity, as limiting the search space has proved itself as an effective method to increase the efficiency of model verification tools.
- **Chapter 5** presents the first technique for improving the efficiency of verification tools. Given a correct model and a series of modifications, it consists in determining which parts of the resulting model may impact model correctness. The intent is to limit the verification analysis to those parts, omitting the rest of the model in the process. We call it incremental verification of models.
- **Chapter 6** presents our second technique for improving the efficiency of verification tools. In this case, the technique addresses one of the most important flaws of bounded verification approaches: manually setting the search space boundaries. The technique presented in this chapter analyzes the model to be verified to try to automatically determine what are the appropriate boundaries for the search space. This chapter, along with Chapters 3, 4 and 5 make up the second part of this thesis, devoted to verification tools, and techniques to improve their effectiveness.
- **Chapter 7** presents the state of the art in the field of model transformation testing, describing the existing approaches and their weaknesses. Some improvement proposals are also suggested.
- **Chapter 8** introduces a tool for the generation of input test data for model transformation testing, based on the analysis of model transformation internals. This tool is based on the EMFtoCSP tool presented in Chapter 4, and is the first proposal on how static model verification tools can be of assistance to test model transformations.
- **Chapter 9** introduces a second tool, also for the generation of input test data for model transformation testing, but in this case, it is based on the analysis of the model transformation specification. As it was the case for the tool of Chapter 8, this tool also relies on EMFtoCSP. It is the second proposal showing how to use static model verification tools to assist in the process of testing a model transformation. This chapter, along with Chapters 7 and 8 constitute the third part of this thesis, oriented to discuss the role static model verification tools may play in model transformation testing.
- **Chapter 10** draws the conclusions of this thesis and ideas for future research. It constitutes in itself the last part of this document.

1.4 List of Publications

Several parts of the work conducting to this thesis have been published in the following papers (listed in reversed chronological order)

1. Carlos A. González and Jordi Cabot, Test Data Generation for Model Transformations Combining Partition and Constraint Analysis, 7th Int. Conf. on Model Transformations (ICMT), Springer, Lecture Notes in Computer Science, volume 8568, pp. 25 - 41, 2014.
2. Carlos A. González and Jordi Cabot, Formal verification of static software models in MDE: A systematic review, Elsevier Information and Software Technology Journal, Volume 56, Number 8, pp. 821 - 838, 2014.
3. Carlos A. González and Jordi Cabot, ATLTest: A White-Box Test Generation Approach for ATL Transformations 15th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS) Springer, Lecture Notes in Computer Science, volume 7590, pp. 449 - 464, 2012.
4. Carlos A. González and Fabian Büttner and Robert Clarisó and Jordi Cabot, EMFtoCSP: A tool for the lightweight verification of EMF models, Proceedings of the workshop "Formal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA" , pp. 44 - 50, 2012.

2

Basic Concepts

This chapter introduces some of the basic terminology used throughout this thesis, with the intent of facilitating the contextualization of the work presented in Parts II and III. In particular, it gives a brief overview of the concepts of Model-Driven Engineering (MDE), software verification, software validation and constraint programming.

2.1 Model-Driven Engineering

One of the most powerful weapons in the arsenal of the software engineer is the ability to abstract complexity away. Abstraction facilitates reasoning about complex real-world phenomena, while retaining only the information that is relevant for a particular purpose, and hiding away nonessential details. When these abstractions are expressed textually, graphically, or as a combination of both, it is typical to refer to them with the word "model". Models are important elements of all software engineering activities, but in many cases, they play a secondary role, serving as documentation, or somehow tangentially supporting the software development process.

Model Driven Engineering (MDE in short) is a methodology that seeks to change this, by promoting the utilization of models as first-class citizens in all software engineering activities. As stated in [3], the role of MDE consists in defining sound engineering approaches to the definition of models, to their modification (that in the MDE terminology is known as "model transformation"), and their integration within software engineering activities. An immediate consequence of this is that, in MDE, the creation of models goes beyond the process of informally depicting ideas

or real-word phenomena. In MDE, models must feature precise and well-defined syntax and semantics, so that they can be automatically interpreted by a computer.

2.1.1 Modeling and Metamodeling

Models and model transformations must be expressed using some kind of notation, which in MDE receives the name of "modeling language". In reality, in MDE, everything is a model. An immediate implication of this is that these modeling languages are nothing but models, and therefore as such, they are also expressed using a graphical representation, text, or both combined, and must feature well-defined syntax and semantics. The utilization of models to describe models is known as "metamodeling". If models are an abstraction of some real-world phenomena, metamodels are abstractions describing the properties of models themselves. Metamodels therefore define a modeling language. It is said, that the models described by a metamodel conform to that metamodel, pretty much the same way that a given program written in a certain programming language, conforms to the grammar of that programming language.

Metamodeling is a recursive process. As a metamodel defines the modeling language that describes the whole class of models represented by that language, a meta-metamodel defines the modeling language that describes metamodels. A popular way of depicting this is by using the four-layered architecture diagram (Fig. 2.1), proposed by the Object Management Group [1] (OMG) within the Model-Driven Architecture (MDA) framework [2]. The MDA framework is the particular vision of the OMG about MDE when applied to the development of software.

The layers in Fig. 2.1 receive the names M0, M1, M2 and M3, respectively. The layer M0 represents the running system where the real-world objects, or instances exist. These instances conform to the models in layer M1, which in their turn, conform to the metamodels in layer M2. That is, models in M1 are described by the modeling languages in M2. Along with this line of reasoning, metamodels in M2 conform to the meta-metamodel in M3, which defines the concepts used in M2, and also the ones used in M3 itself. In this regard, OMG proposes an standard meta-metamodel for M3 called Meta-Object Facility (MOF) [3].

2.1.2 Model Classification

Models can be classified in multiple ways, but when it comes to MDE, the two most popular ways of doing this is by considering the part of the system, where the model puts the focus on, or the scope of the modeling language.

1. http://www.omg.org/
2. http://www.omg.org/mda/
3. http://www.omg.org/mof/

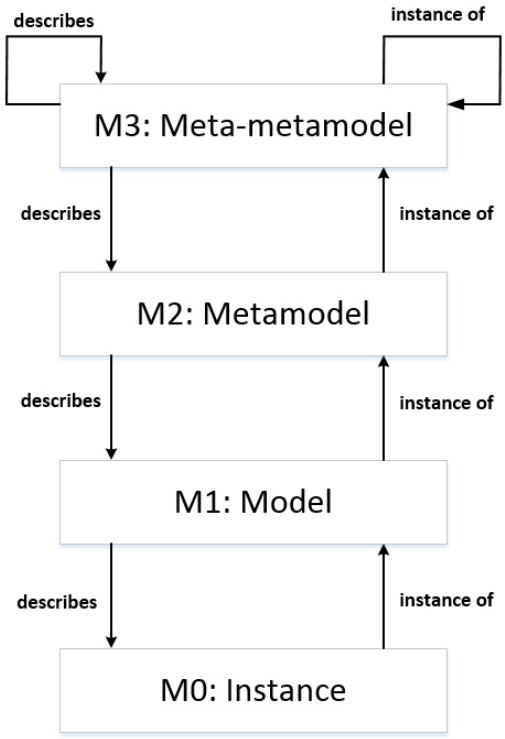


Figure 2.1: MDA 4-layer architecture

Regarding the first way, when following an MDE approach, it is typical to build different models to gather different aspects of the system under analysis. In these cases, it is quite common to distinguish among two types of models: static or structural models and dynamic or behavioral models.

Static models are those models used to represent, totally or partially, the structure and architecture of the system. These models show, in general, a time independent view of the system. An example of this type of model is the UML [4] (Unified Modeling Language) class diagram. Dynamic models, on the other hand, describe the behavior of the system, which may include among other aspects, the description of control or data flows within the system, how the internal state of the different components of the system evolves throughout time, or even the way the system must be used by its users. Some examples of behavioral models are UML sequence diagrams or UML activity diagrams.

Finally, and regarding the scope of the modeling language, it is possible to distinguish two types of modeling languages: Domain Specific Modeling Languages (DSMLs) or General Purpose Modeling Languages (GMLs). DSMLs are designed with the intent of facilitating the creation of models describing the reality of a particular domain or context of interest. GMLs, on the other hand, are modeling languages that can be used to describe models in any domain. The most typical example of GML is UML.

4. http://www.omg.org/spec/UML/

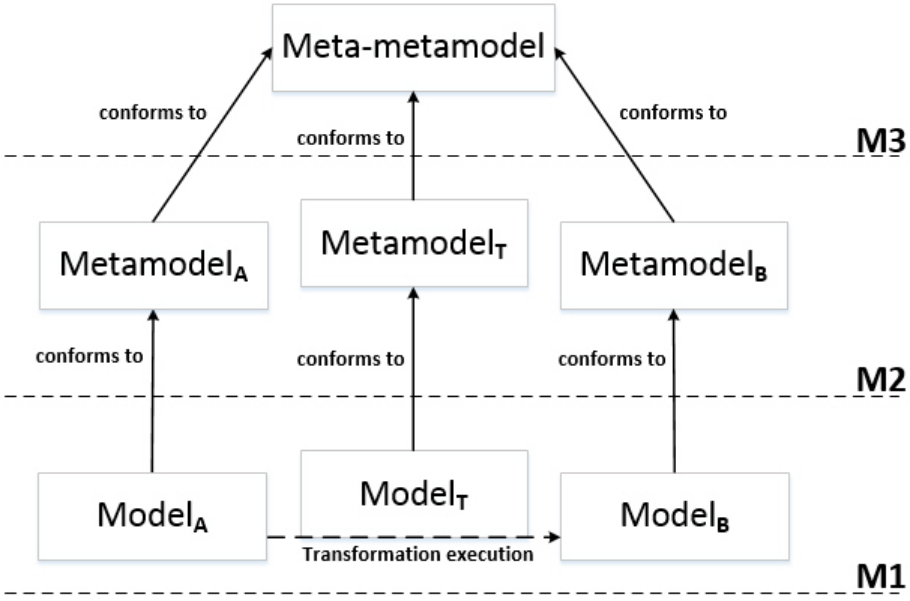


Figure 2.2: Relationship between models and model transformations

2.1.3 Model Transformations

Model transformations are the other key element to MDE. In general terms, they can be seen as a mechanism to transform a model described in a source language into a model described in a target language. That is, model transformations define the rules by which the constructs from the source language are transformed into constructs from the target language.

Fig. 2.2 shows the relationship between models, metamodels and model transformations. It can be seen that the model transformation conforms to some kind of model transformation language. This stands to reason, since, as it was mentioned before, everything in MDE is a model. The model transformation language is nothing but a metamodel, defined in M2 as any other metamodel, that conforms to the meta-metamodel in M3. It can also be seen in the figure, that model transformations are applied at the modeling level (M1), which makes sense, since are models and not model instances what it is transformed.

Model transformations, as it is the case with models, can be classified in multiple ways. At the top level, it can be distinguished between model-to-model and model-to-text transformations [5]. In the first ones the target is a model, and in the second ones, it is just a set of text strings, like source code or documents. Model-to-model transformations, in their turn, can also be object of multiple classifications. Attending to the source and target metamodels, for example, transformations can be endogenous, when both metamodels are the same, or exogenous, otherwise. If the level of abstraction of the source and target models is considered, it would be possible to talk about horizontal transformations, when the level of abstraction is the same, or vertical transformations, when it differs [6]. Finally, when con-

sidering the way the transformation is specified, it is possible to talk about operational (imperative), declarative and hybrid approaches. Operational approaches, like Query/View/Transformation (QVT) Operational mappings [5], focus on how the transformation itself must be performed, whereas declarative approaches, like QVT Relations [6], focus on the relation between source and target metamodels. Finally, hybrid approaches, like the ATL Transformation Language (ATL) [7], combine both. ATL will be described in more detail in Part III, when talking about testing of model transformations.

2.1.4 The Object Constraint Language (OCL)

The Object Constraint Language (OCL) [7] was created with the intent to overcome the limitations shown by UML at the time of precisely specifying detailed aspects of a system design [8]. With the passing of time, OCL has become a key player to MDE and, as of now, it is widely used not only to express integrity constraints, but also to describe operation contracts or query operations over models; or even to facilitate the description of model transformations or code generation templates, among many other uses.

OCL is a textual, typed, and mostly declarative language. This means that any OCL expression evaluates to a type, and that the presence of imperative statements, like for example assignments, is scarce. One of the most important characteristics of OCL is that it is side-effect free, therefore, OCL expressions do not modify the state of the system.

Due to its extensive use, the analysis of OCL expressions is an integral part of those mechanisms devoted to increase the reliability of MDE approaches, and the techniques presented in this document are not an exception. More details about OCL will be provided in subsequent chapters when needed.

2.2 Software Verification

Software Quality Assurance (SQA) comprises all activities required to make sure that a software product meets certain quality objectives [9]. It is difficult to come up with a precise definition of quality, though. After all, quality means different things to different people, and factors like the domain or the point of view are of great influence here [10, 11]. To the best of our knowledge, McCall et al. [12] where the first ones in describing software quality in terms of quality factors and quality criteria. Even though with the passing of time, a lot of quality factors have been

5. http://www.omg.org/spec/QVT/1.1/
6. http://www.omg.org/spec/QVT/1.1/
7. http://www.omg.org/spec/OCL/2.4/

identified (efficiency, reusability, maintainability, etc) and different quality models have been proposed (ISO/IEC 9126[8], ISO/IEC 25010[9], CMMI[10]), as of now, it is generally accepted that the most important quality factors are those impacting software reliability.

Software verification makes reference to certain procedures and techniques devoted to identify and solve software problems, that is, to preserve software reliability. The following definition of "software verification" can be found in the IEEE Standard Glossary of Software Engineering:

> *"Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase"* [13].

However, it is more typical to describe "software verification" by using the definition proposed by Barry W. Boehm:

> *"Am I building the product right?"* [14].

It is also quite common to narrow down the meaning of the term "verification", and use it in the sense of "formal verification". Formal verification is about the utilization of formal methods to try to prove software correctness. The term formal methods, as stated in [15] refers to the use of mathematical modeling, calculation and prediction in the specification, design, analysis and assurance of computer systems and software. The reason why it is called formal methods is to highlight the character of the mathematics involved.

2.2.1 Formal Methods and Software Correctness

The utilization of formal methods to try to ensure software correctness can be seen, roughly speaking, as a three-step process:

- Building a formal specification of the system under analysis
- Reasoning over the specification to prove properties about it
- Obtain, from the formal specification, an implementation of the system

Checking software correctness by using, either formal methods, or any other technique, requires the presence of a specification. A specification is, in a broad sense, a description of what the system does and therefore, serves as the frame of reference for all the techniques devoted to try to ensure software correctness. In general, a specification needs to be as precise and unambiguous as possible, but this is especially important when dealing with formal methods. Normally, this is

8. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749
9. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733
10. http://www.cmmiinstitute.com

achieved by using a specification formalism featuring a well-defined syntax and precise semantics. In these cases, it is typical to speak of formal specifications.

Finding the right formalism to fit the specification needs is not an easy task. Actually, there is no shortage of formalisms available, although some of them might be more adequate than others, depending on the perspective followed to build the specification. For example, formalisms like Abstract State Machines (ASM) [16], the B language [17], the Z notation [18], or even automatas [19, 20] seem to be a good choice to describe the system as a set of states with system operations being expressed as transitions from one state to another. Alternatively, logic programming languages, such as Prolog [21], functional languages based on λ-calculus [22], such as Haskell [23], or rewriting systems [24], might be adequate to build a specification focusing on the manipulated data, how they evolve, or the way in which they are related.

Once the specification has been built, the next logical step is to prove properties about it. Reasoning about the specification can be done manually, or with the help of computer-based tools. In this last case, the election of the tools to be used will also depend on the formalism employed to build the specification. For example, in the case the specification represents the system as a set of states, model checking [25] could be the right technique to use. On the other hand, if an algebraic formalism is employed, and depending on its expressiveness, using tools like SMT solvers [26, 27] or interactive theorem provers [28, 29] could be more adequate.

After proving that a specification features the desired properties, it is the time of trying to obtain a matching implementation. As in the previous steps, there are different alternatives that may be followed. In some cases, the specification is a program that can be executed directly. This is the case of specifications expressed, for example, with logic-based languages like Prolog or functional languages like Haskell, that offer scope for both specification and implementation in the same language. If the specification is not executable, then the implementation must be derived from the specification, which, in its turn, implies the verification of the derivation mechanism employed. These approaches are commonly referred to as "correct-by-construction" software development.

Finally, it is worth mentioning that formal methods can be applied in varying degrees, and not necessarily over all the components of the system under analysis. Normally, the level of formalization applied is influenced by factors like the nature of the system or budgetary issues, since the cost of full verification may be prohibitive. Some authors [30] advocate for a light utilization of formal methods in order to facilitate their adoption by the industry. This is known as "lightweight formal methods".

2.3 Software Validation

Verification techniques are not the only ones available when it comes to preserve software reliability. There is another important group of techniques that fall under the umbrella of what is called "software validation".

There are multiple definitions of what "software validation" means. For example, the following definition can be found in the IEEE Standard Glossary of Software Engineering:

> *"Validation is the process of evaluating software during or at the end of the software development process to ensure compliance with software requirements"* [13].

As it was the case for software verification, Barry W. Boehm also came up with a definition for "software validation" that has become very popular over time:

> *"Am I building the right product?"* [14].

Software validation therefore, like software verification, also makes reference to certain procedures and techniques aimed at trying to identify and solve software problems. When compared to software verification, software validation can be regarded as a "cheaper" way of establishing a certain degree of confidence over the correctness of a piece of software, and therefore it is especially useful in those scenarios where software verification techniques cannot be applied, or the cost of their application is too high. Software validation is typically conducted by means of testing techniques, which will be the focus of the next subsection.

2.3.1 Software Testing

Software testing, also known as program testing, can be viewed as the destructive process of trying to find the errors (whose presence is assumed) in a program or piece of software, of course, with the intent of establishing some degree of confidence that the program does what it is expected to do [31]. Considering testing a destructive activity has to do with the fact that its goal is not to prove software correctness, but uncover the presence of errors. As Dijkstra stated in [32], testing can be used to show the presence of bugs, but never to show their absence. There are a plethora of testing techniques that can be used at different stages during the software life cycle, but they all can be classified into two big groups:

– Static analysis techniques.
– Dynamic analysis techniques.

Static analysis techniques [33], that some authors also called "verification testing" [34], are those techniques devoted to the examination of the program code

with the intent of reasoning about the behavior that the software is going to exhibit at runtime. This can be complemented with the examination of other documents, such as design models or requirements documents, if necessary. The main characteristic of this type of activities is that they are conducted manually and therefore, do not require code execution to uncover errors. Some techniques that fall into this category are code inspections or walkthroughs [31].

Dynamic analysis techniques [33], on the other hand, do require software execution to expose the presence of bugs. This typically implies following a methodology that can be summarized in the following three points, which are repeated a certain number of times, until a specific stopping condition is met.

– Determine the adequate input data to test the program with.
– Run the program with the input data obtained from the first stage.
– Analyze the outputs yielded by the program with the intent of uncover errors.

The information needed to conduct these three steps is typically bundled into an item called "test case". This information includes, at least, the input data values the program is going to be run with; any other execution condition required by the test case, like for example a specific hardware configuration or a certain database state; and the expected outputs to be produced during the program execution. In relation to this last point, the expected outputs are paramount, since without them, it is impossible to determine whether the execution of the test case uncovered any errors. In testing terminology, it is usual to use the word "oracle" [35] to make reference to that program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object. An oracle can be as simple as a manual inspection or as complex as a separate piece of software.

It is generally accepted that the more test cases are created and the more time is spent running the software, the higher is the probability of finding errors and therefore ending up with a more reliable software. However, since the number of test cases that can be created to test a piece of software is potentially infinite, it is necessary to establish some strategy to carry out testing in an effective way.

Two of the most prevalent strategies are black-box testing and white-box testing. The main difference between the two is the kind of information considered to generate test cases, the rest of the testing process being essentially the same.

When following a black-box approach, the tester does not consider any information related to the internal structure of the system under test. She "knows" what it does, but not how it does it. It is a black-box, hence its name. Because of this, the creation of test cases by means of black-box approaches is usually based on information gathered from the software specification, requirements documents or some domain knowledge the tester might have. In light of this, black-box approaches are quite adequate to uncover errors related to software requirements or the software specification.

White-box approaches, on the other hand, focus on the internal structure of the software to be tested, and therefore, the presence of the source code is required. When the tester applies a white-box strategy, the test case generation process is driven by information such as the presence of branches or loops in the software code, or more generally, the control flows and data flows that can be inferred from the analysis of that code. All this makes white-box approaches especially suitable for the discovery of logic errors.

In spite of the variations among the different strategies available, it is generally accepted that generating test cases through a combination of distinct testing strategies, maximizes the probability of uncovering errors and therefore, the success of the testing experience.

It is also important to mention that testing techniques can be applied in varying degrees and at different stages during the software life cycle. In this regard, it is typical to talk about four different "levels of testing" [36], namely unit testing, integration testing, system testing, and acceptance testing. Unit testing makes reference to the evaluation of system components in an isolated manner to detect functional or structural defects. Integration testing focuses on the evaluation of the interaction among different parts of the system, so the focus usually spreads over several components somehow related to each other. System testing checks the system as a whole, not only to determine the presence of defects, but also to evaluate quality aspects like performance, usability, security or the documentation, among others. The last testing level, acceptance testing has to do with ensuring that the system satisfies the initial requirements. and therefore meets the needs of its end users. It is important to remark that these "levels of testing", although popular, are not the only types of testing available. Actually, they do not even cover all the stages that typically occur during the software life cycle. For example, another type of testing known as regression testing, may be especially useful during the software maintenance stage.

Finally, it is worth noting that there is not a preferred or standard way to integrate testing techniques within the rest of activities that take place during the software development life cycle. However, a popular software development paradigm like Test Driven Development (TDD) [37] may facilitate this.

2.4 Constraint Programming

Constraint programming, as stated in [38], is the study of computational systems based on constraints. The idea behind constraint programming is to solve problems by stating requirements (constraints) about the problem area, and then finding a solution satisfying all these constraints. As described in [39], in its most basic form, constraint programing consists in finding a value for each one of a set of

problem variables where constraints specify that some subsets of values cannot be used together. The problems addressed by Constraint Programming are known as Constraint Satisfaction Problems (CSP).

Informally speaking, a constraint can be viewed as a relation between variables and the values they can take. The term "constraint" has been typically used by physicists and mathematicians for a long time, but when it comes to computer science, though, it was not until the early sixties that Ivan Edward Sutherland adopted it for the first time, while working on the Sketchpad project [11], in the area of computer graphics. In particular, he proposed the following definition, that can be found in his PhD thesis:

> *A constraint is "a specific storage representation of a relationship between variables which limits the freedom of the variables, i.e., reduces the number of degrees of freedom of the system"* [40]

After this pioneering work, the earliest ideas leading to the concept of constraint programming came from the field of Artificial Intelligence (AI) in the sixties and seventies. One of the major breakthroughs was the formalization of a CSP for the first time, which was made by David L. Waltz [41]. Formally speaking, a CSP can be represented as the tuple $CSP = \langle V, D, C \rangle$ where V denotes the set of variables, D the set of domains, one for each variable, and C the set of constraints. It is typical to describe the constraints in a CSP by means of a combination of arithmetic expressions, mathematical comparison operators and logical operators.

A solution to a CSP is an assignment of values to the variables such that all constraints are satisfied. As described in [42], there are different techniques that can be used for solving a CSP, namely: domain specific methods, general methods or a combination of both. The expression "domain specific methods" makes reference to those implementations of special purpose algorithms, typically provided in the form of libraries, like for example, algorithms for the resolution of systems of equations, or linear programming algorithms. On the other hand, "general methods" has to do with those techniques to reduce the search space where to look for a solution (which are commonly referred to as constraint propagation techniques); and with specific search methods, such as backtracking or branch and bound search. For efficiency reasons, whenever possible, domain specific methods should prevail over general methods. From an architectural standpoint, the tools used to solve CSPs are typically developed as frameworks, with built-ins supporting different search methods and constraint propagation techniques, accompanied by domain specific methods in the form of libraries, often called constraint solvers.

When it comes to the actual resolution of CSPs, these tools attempt to assign values to variables following a certain order. If the partial solution violates any con-

11. http://en.wikipedia.org/wiki/Sketchpad

straint, then the last assignment is reconsidered by either trying a new value in the domain, or backtracking to previous variables, if there are no more values available. The utilization of constraint propagation techniques to identify unfeasible values in the domain of unassigned variables, helps to speed up the process by pruning the search tree. The process continues until a solution is found or all possible assignments are considered. In this second case, the CSP is called unfeasible. What can be followed from all this, is that in order to ensure termination, the domains of the variables must be finite.

Finally, it is also important to mention, that in order to solve a CSP with computer-based tools, the CSP must be expressed using some kind of notation. Popular alternatives here are the utilization of object-oriented/procedural languages or declarative notations. In the first case, well known languages like C++, Java or .NET can be used, as for example, it is the case with the IBM CPLEX CP Optimizer [12]. In the second case, the Constraint Logic Programming (CLP) paradigm proposed by Jaffar and Lassez [43], which merges constraint solving and logic programming is especially popular. An example of a tool which allows representing and solving CSPs using this paradigm is the ECL^{i}PS^e Constraint Programming System [13], which is at the heart of some of the contributions presented in this thesis.

2.5 Summary

With this chapter finishes the introductory part of this thesis. The objectives have been presented, and some basic terminology has been introduced. In the next part, the focus will be on the first contributions of this doctorate, the ones devoted to improving the landscape of static model verification tools.

12. http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/
13. http://eclipseclp.org

Static Model Verification

3

Landscape of Static Model Verification Approaches

After introducing the basic terminology, in Part II the focus is on the discovery of techniques to increase the efficiency of tools devoted to the verification of static models. Static models are important because, as it was mentioned in previous chapters, are possibly the models more commonly used when specifying a software system [4]. A classical example of static model is the UML class diagram so, from now on, it will be common to make reference to class diagrams constructs, such as classes, associations or OCL constraints.

In this first chapter of Part II, the target is to depict what the current landscape regarding static model verification is. We will start by stating some generalities about the matter and after that, we will describe what are, as of this writing, the existing approaches in the field. To finish the chapter, we will discuss certain aspects that can be improved. How to address some of them will be the objective of the rest of chapters in this part.

3.1 Generalities About Static Model Verification

Formal verification of static models makes reference to those approaches devoted to proving model correctness by means of the utilization of formal methods and formal analysis techniques. However, and although some may find it surprising, there is no a universal definition of what "model correctness" means. On the contrary, there are many ways a given model may be considered "correct". Because of this, it is typical to refer to model correctness as the ability of the model under

analysis to satisfy one or more correctness properties. These properties state certain characteristics the model must feature in order to be considered correct.

In general terms, it is possible to classify correctness properties into two big groups: the group of properties about the instantiability of the model and the group of properties about the relationship among constraints. Regarding the properties that fall into the first group, the most important one is called "satisfiability". A model is consider "satisfiable" when it is possible to create instances out of it. As we will see in the following section, some of the studies analyzed distinguish among different "flavors" of satisfiability: for example, strong satisfiability means that the legal model instance must include instances of all the classes and associations in the model, and weak satisfiability, being less strict, does not enforce the instantiation of all classes and associations. Another property belonging to this first group is "liveliness of a class", that is to check whether it is possible to create legal instances of the model including at least one instance of a certain class. This property can be regarded as a particular case of satisfiability. In general, satisfiability is of special importance because the rest of correctness properties can be expressed in terms of this property. When it comes to the second group of correctness properties, some of the existing verification approaches are able to check the presence of redundant constraints in the model, some others can check whether two given constraints in the model contradict to each other, or even if some constraints are subsumed by others.

When it comes to the way verification tools in this field work, the verification process is not very different than the general approach described in Chapter 2. Verification of static models can be divided in two different stages. In the first one, the formalization takes place, that is, the model along with the correctness properties to be checked are somehow represented in the formalism of choice. With the formal representation in place, the second stage takes over. It consists in reasoning over that formalism, usually with the help of some specialized tool, to see whether the correctness properties are satisfied or not.

The completeness and degree of automation of this reasoning process is strongly dependent on the degree of support given to the OCL. This is because supporting OCL in its full generality leads to undecidability issues, as stated in the work of Berardi et al. [44]. In this scenario, verification approaches that are complete are also user-interactive (i.e. they need help from the user to steer the verification process). This can be problematic since it usually requires from users a non-negligible expertise in the formalisms employed. Because of this, the majority of approaches supporting OCL in its full generality are automatic and ensure termination, but this is achieved at the expense of its completeness. This is done by following a bounded verification approach, in which users typically have to configure beforehand certain parameters to drive the reasoning process, but once it is launched, user intervention

is not required. In particular, it is typical of these approaches to put in the users' hands the responsibility of setting the search space boundaries where to look for a solution of the problem. In this scenario, results are only conclusive if the model is found to be correct. Nothing can be concluded otherwise. It might be that the model is incorrect, or that it is correct, but the solution to the problem lies outside the search space considered. On the other hand, approaches supporting only a subset of OCL are automatic and complete since they are not affected by the undecidability issues stated in [44]. Finally, and before presenting the existing verification approaches it is worth mentioning that in the majority of cases when a model is found to be correct, these tools yield a valid instance of that model as a proof of correctness.

3.2 Static Model Verification Approaches

While trying to fulfill the objectives of this thesis, we came across to an important number of scientific papers that, one way or another can be directly related to the formal verification of static models. Some of these works put the focus on formalization aspects, some others on theoretical aspects regarding the reasoning stage, and finally some others present verification tools. In all cases, these works are part of coarse-grained studies where each paper contributes to a specific part of the verification problem. In particular, we have identified up to 17 different studies that are summarized in Table 3.1. In what follows, we present these studies and their main characteristics. To facilitate their contextualization, we also add citations to other relevant works, whenever necessary. In those studies complemented with the presence of a verification tool, we expand the description to also provide a brief description of the tool itself. Finally, Tables 3.2, 3.3 and 3.4 summarize our findings to facilitate the reader the comparison of these studies attending multiple factors.

The first study (S1) comprises the works directly related to UMLtoCSP. UMLtoCSP, developed by Cabot et al. [45,46], is a Java tool for the formal verification of UML/OCL models based on constraint programming. The tool works by translating a class diagram, its OCL constraints and the desired verification properties into a CSP. The data input interface is a bit limited. The class diagram must be expressed in an XMI format created with ArgoUML [1], and the OCL constraints must be provided in a separate text file. In general, the GUI looks outdated. The CSP built by the tool is expressed as a Constraint Logic Program (CLP), this is because, for the resolution of the CSP, the tool relies on the ECL^{i}PS^{e} Constraint Programming System [2], which requires CSPs to be expressed in this notation.

1. http://argouml.tigris.org/
2. http://eclipseclp.org

Table 3.1: Summary of the 17 studies identified

Study	Representative Name	References
S1	UMLtoCSP	[45–47]
S2	OCL2FOL	[48]
S3	FiniteSat	[49, 50]
S4	AuRUS	[51–55]
S5	DL	[44, 56–65]
S6	OCL-Lite	[66, 67]
S7	OODB	[68, 69]
S8	HOL-OCL	[70–72]
S9	UML2Alloy	[73, 74]
S10	USE	[75–81]
S11	BV-BSAT	[82, 83]
S12	PVS	[84]
S13	KeY	[85]
S14	Object-Z	[86]
S15	UML-B	[87]
S16	CDOCL-HOL	[88]
S17	MathForm	[89]

Table 3.2: Formalization techniques used in each study

Study	Formalization Technique
S1 (UMLtoCSP)	CSP
S2 (OCL2FOL)	FOL
S3 (FiniteSat)	System of Linear Inequalities
S4 (AuRUS)	FOL
S5 (DL)	Description Logics, CSP
S6 (OCL-Lite)	Description Logics
S7 (OODB)	TQL^{++}
S8 (HOL-OCL)	HOL
S9 (UML2Alloy)	Relational Logic
S10 (USE)	Relational Logic
S11 (BV-BSAT)	Bit-vector Logic
S12 (PVS)	HOL
S13 (KeY)	Dynamic Logic
S14 (Object-Z)	Object-Z
S15 (UML-B)	B
S16 (CDOCL-HOL)	HOL
S17 (MathForm)	Mathematical Notation

Although UMLtoCSP was designed with the intent of supporting OCL constraints in its full generality, as of this writing, not all the OCL constructs are supported. Even though, the tool features some notable characteristics: user intervention during the reasoning process is not required (i.e. it is automatic) and termination is ensured. This is possible because the tool follows a bounded verification

approach, which also means that, as typical of these approaches, the verification process is not complete. Regarding correctness properties, UMLtoCSP supports the verification of strong satisfiability, weak satisfiability, liveliness of a class, lack of constraints subsumption and lack of constraint redundancies. If the verification process succeeds, the tool presents an image of a valid model instance as a proof. To do this, it requires the presence of the graph visualization package Graphviz [3].

Unfortunately, the last version available (June 2009) presents several bugs and 64-bit platforms or modern operating systems like Windows 7 or Windows 8 are not supported. Besides, it is not unlikely for the verification process to take quite some time in many occasions. The source code is not available on the website.

UMLtoCSP was later on complemented with the work of Shaikh et al. [47] consisting in the development of a slicing technique for UML/OCL class diagrams. The presence of this technique turned UMLtoCSP into a more efficient tool when verifying weak satisfiability or strong satisfiability.

The second study (S2) is about the work of Clavel et al. [48] on formalizing and reasoning over OCL constraints. In this work, a mapping from a subset of OCL into FOL is proposed with the intent of supporting verification using automated reasoning tools like Prover9 [4], an automated theorem solver, and Yices [5], a SMT solver. In particular, the authors propose reasoning on their own notion of (unbounded) unsatisfiability of OCL constraints over a class diagram.

The third study (S3) collects the works of Azzam Maraee and Mira Balaban, who developed a linear programming based method for reasoning about finite satisfiability of UML class diagrams with constrained generalization sets [49]. In the authors' words, finite satisfiability is the problem of deciding whether a given class has a finite, non-empty extension in some model. Their method builds on top of the work of Lenzerini and Nobili [90], which is based on the transformation of the cardinality constraints into a set of linear inequalities whose size is polynomial in the size of the diagram. This way, the finite satisfiability problem is reduced to the problem of finding a solution to a system of linear inequalities. The algorithm proposed, called "FiniteSat", was later on improved [50] to handle all the types of constraints included in an enhanced version of the Description Logics to class diagrams translation presented in [44].

The fourth study (S4) congregates the verification works based on the CQC Method [91], a mechanism to perform query containment tests on deductive database schemas, that has also been used to determine properties like satisfiability or predicate liveliness over this type of schema. In this regard, Queralt et al. presented AuRUS [51], a tool for assessing the semantic quality of a conceptual schema consist-

3. http://www.graphviz.org/
4. http://www.cs.unm.edu/ mccune/mace4/
5. http://yices.csl.sri.com/

ing in a UML class diagram complemented with OCL arbitrary constraints, which extends SVTe [92], a relational database schema validation tool. Apart from the satisfiability of the conceptual schema, AuRUS can verify liveliness of classes or associations and redundancy of constraints, without requiring user intervention during the reasoning process. The tool works [52] by first translating both, the class diagram and the OCL constraints into a set of first-order formulas that represent the structural schema, and then verifying, by using the CQC Method, whether the supported properties hold. In the case that the properties do not hold, the tool is able to give the user a hint about the changes of the schema that are needed to fix the problem identified [55]. AuRUS does not guarantee termination when dealing with general OCL expressions, but it does so [53] when dealing with a specific subset of constraints.

Finally, in [54] the authors presented several improvements, like an enhanced version of the conceptual schema to logic translation, or refinements on the mechanism presented in [53] which is used by AuRUS to determine whether the reasoning process will terminate or not.

The fifth study (S5) compiles the work developed by Calvanese et al. on reasoning over entity-relationship models and UML class diagrams since the year 2002. Related to this, Cadoli et al. [56–58] developed an approach to encode the problem of finite model reasoning (i.e. checking whether a class is forced to have either zero or infinitely many objects) in UML class diagrams as a CSP that is solved by relying on the use of off-the-shelf tools for constraint modeling and programming. These works exploit the encoding of class diagrams in terms of Description Logics proposed by Berardi et al. [44, 63, 64] and Cali et al. [65], to take advantage of the finite model reasoning techniques developed for Description Logics in [93], based on reducing the problem of reasoning on a Description Logics knowledge base to the problem of finding a solution for a set of linear inequalities.

Moreover, the work of Berardi et al. [44, 63] has also served as a basis for the complexity analyses conducted by Artale et al. [59–62] which have established important results about the problem of verifying full satisfiability over different variants of UML class diagrams or entity-relationship models.

The sixth study (S6) contains the work related to OCL-Lite [66, 67], a fragment of OCL that ensures termination and completeness when reasoning on UML conceptual schemas enriched with arbitrary constraints within the bounds of this fragment. Apart from the identification of such a fragment, the authors propose an encoding of UML class diagrams enriched with constraints within its bounds in Description Logics. In this regard, they take advantage of the works developed by Calvanese et al. that have been described in [S5]. Finally, they show how it is possible to use existing reasoners to provide reasoning support to check properties like schema satisfiability or constraint redundancy over these models.

The seventh study (S7) refers to the work of Anna Formica on checking finite satisfiability of database constraints. In particular, a decidable graph-theoretic approach to finite satisfiability checking is proposed in [68]. This approach, which is limited to integrity constraints involving comparison operators, was later on expanded in [69] to cover cardinality constraints among others. In both cases, the database schemas are described using fragments of TQL^{++} [94], an object-oriented data definition language aimed at modeling the structural aspects and integrity constraints of object-oriented database models [95].

The eighth study (S8) is about the formal proof environment HOL-OCL [70], developed by Brucker and Wolff. HOL-OCL is an interactive proof environment that can be used to analyze UML/OCL models created with ArgoUML. It has been integrated into a framework supporting a formal model-driven engineering process, which is described in [71]. HOL-OCL works by automatically encoding the class diagram along with the OCL constraints in Higher-Order Logics (HOL). This encoding, which is described in detail in [72], can then be used to reason over the model by means of the interactive theorem prover Isabelle [96]. A drawback of HOL-OCL is that it is a tool extremely hard to use for the user not familiarized with formal methods and Isabelle. Not even its installation is trivial, since it presents an important number of prerequisites. Regarding the verification stage is, in general, interactive, and requires building Isabelle theories as part of the process. Last version is 0.9.0 (the year is not indicated) and the downloadable package includes the source code.

The ninth study (S9) makes reference to the works about UML2Alloy [73, 74], which is the name of a Java standalone application developed by Anastakasis et al., that can be used to check the satisfiability of a UML class diagram enriched or not with OCL constraints. UML2Alloy, as it can be inferred from its name, works by transforming the model to be verified into the relational logic of Alloy [6] [97], which is then fed into the SAT solvers embedded within the Alloy Analyzer (bundled with the tool). As UMLtoCSP, the GUI looks outdated. The different steps are distributed in tabs. Once the model is loaded, it is necessary to set the boundaries of the search space and determine how OCL statements will be transformed into Alloy. Regarding the verification process in itself, UML2Alloy, as UMLtoCSP, follows a bounded verification approach (i.e. it is not complete). If it succeeds, the tool presents a valid model instance as a proof. This instance can then be exported as a PNG image or a PDF document. a drawback of this tool is that it is a bit unintuitive, forcing users to run actions like parsing XMI files or transform the input data to Alloy, explicitly. Last version is 0.52 Beta, built on May 2009. The source code was not available on the website.

The tenth study (S10) relates to the work of Gogolla et al. around the USE

6. http://alloy.mit.edu/alloy/

tool [75, 76]. USE is a Java standalone tool that, although originally conceived as a validation tool, has evolved significantly throughout the years and now, since version 3.0.5, can be used to verify class diagrams enriched or not with OCL constraints. The tool has a long history behind, since version 0.1 was created in 1999 and, compared to the rest of tools analyzed, it is probably the most polished one.

In one of its initial versions [75], the tool worked by generating instances, or fragments of instances of a given input model (these instances or fragments are called snapshots in the USE terminology), to be checked, one by one, against a series of OCL invariants. This version of the tool was able to support model verification to a certain extent, like for example, to check constraint independence as shown in [77, 78]. However, since the original approach presented the problem of the enumerative nature of the snapshot generator, it was improved later on by Kuhlmann et al. [79–81] with the development of a mechanism to translate UML and OCL concepts into relational logic, which uses the SAT-based constraint solver KodKod [98] for the reasoning stage. This way, the original snapshot generator was replaced by a more efficient SAT-based bounded search and generation of snapshots fulfilling the user-specified constraints, thus expanding the capabilities of the tool to perform model verification tasks.

Talking in more practical terms, USE verification capabilities are provided in the form of a plug-in called "Model Validator"[7] that, once downloaded and uncompressed, must be copied into the "lib/plugins" directory of the tool. USE reads "USE specification files", that is, text files with ".use" extension where the model along with the OCL constraints are described using a particular syntax. USE only verifies satisfiability. Launching the verification process from the GUI requires a text file with information about the boundaries of the different elements in the specification, as well as which OCL constraints must be taken into account during the process. This file is not required, though, if the process is launched from the USE command-line interface. In this case, the tool will generate one with default values. One of the USE nicest features is that the "Model Validator" plug-in has been designed to support a catalog of SAT solvers, not only KodKod. If the verification process succeeds, the user must open an "Object diagram" window using the GUI, to see the valid model instance provided as a proof. As typical of these tools, USE does not integrate with modeling editing tools. Last version available is 3.0.6 and the downloadable package includes the source code. The source code for the "Model Validator" plug-in is also available in the website.

The eleventh study (S11) compiles the work of Soeken et al. [82, 83] on the verification of UML/OCL models. In their proposed approach a verification problem is encoded into a bit-vector formulation and fed into a SMT/SAT solver for its verification. A verification problem is made up by three different elements: the system

7. http://sourceforge.net/projects/useocl/files/Plugins/ModelValidator/

state, a series of OCL constraints, and the verification task. The system state is an encoding of the attribute assignments in every object as well as of the links between these objects, and the verification task is nothing but the encoding of the property to be verified. Since general OCL constraints are supported, the method follows a bounded verification approach to ensure termination, that is, limits in the number of objects and associations as well as in the domains of the attributes are enforced.

The twelfth study (S12) is limited to the approach proposed by Lukman Ab. Rahim [84] to transform UML class diagrams and OCL constraints into the specification language of the PVS [8] (Prototype Verification System) theorem prover [99]. PVS is based on HOL and comes with a specification language that allows writing theorems to prove a given specification. The approach focuses on describing how a set of rules written using the Epsilon Transformation Language [9] [100] maps UML class diagrams and OCL elements into a proposed PVS metamodel. The idea is to serialize the model into a PVS specification to be fed into the PVS theorem prover for its analysis. It is important to remark that although the mapping proposed does not exclude in advance any type of OCL expression, certain operations in the OCL standard library cannot be mapped due to PVS limitations.

The thirteenth study (S13) refers to the work of Beckert et al. [85] on the formalization of UML class diagrams with OCL constraints into dynamic logic, which is a multi-modal extension of first-order logic (FOL). The approach, that has been implemented in Java and focuses only on the formalization stage, is part of the KeY system [101], a software development tool [10] that seeks the integration of design, implementation, formal specification and formal verification of object-oriented software.

The fourteenth study (S14) relates to the work of Roe et al. [86] on the mapping of UML class diagrams and OCL constraints into a formal specification described with Object-Z [102, 103]. Object-Z is an extension of the Z specification language [104] to facilitate the construction of specifications in an object-oriented style. The mapping, which is described informally, is based on the work of Kim et al. [105], where a formal semantic mapping between the two languages is provided.

The fifteenth study (S15) is about the work of Marcano and Levy [87] describing a systematic translation of UML class diagrams and OCL constraints into a B formal specification. The B specification language [17] provides a means for developing mathematically proven software and systems, through the use of rigorous mathematical reasoning. The approach works by first deducing a B abstract specification from the UML class diagrams, which is then complemented with the addition of a number of B formal expressions generated out of the OCL constraints. The ap-

8. http://pvs.csl.sri.com/
9. http://www.eclipse.org/epsilon/
10. http://www.key-project.org/

proach is supported by the implementation of a prototype tool where the analysis of the generated B specification relies on the utilization of the Atelier-B [11] tool. Although this approach is primarily intended for consistency checking purposes, it can also be used to check verification properties like the presence of contradictions in constraints (incoherent constraints).

The sixteenth study (S16) covers the work of Ali et al. [88] on the formalization of UML class diagrams and OCL constraints into HOL. The work shares some similarities with that of Brucker and Wolff described in [S8], as the intent of reasoning over the resulting encoding using the theorem prover Isabelle [96]. The main difference, though, is the utilization of simpler techniques to build the formalization, so that it can be more accessible to practitioners of the software industry.

The seventeenth study (S17) comprises the work of Marcin Szlenk [89] on the formalization of UML class diagrams into a mathematical notation based mainly on the utilization of sets and partial functions. The formalization is used to outline the subject of reasoning about a class diagram, introducing the formal definition of consistency of a classifier, which is similar in concept to the verification property "liveliness of a class" mentioned in [S1].

11. http://www.atelierb.eu/en/

Table 3.3: Types of model and properties covered in each study

Study	**Model type**	**OCL Constraints**	**Property**
S1 (UMLtoCSP)	Class Diagrams	Yes, General	Strong Satisfiability, Weak Satisfiability, Constraint Subsumption Liveness of a Class, Constraint Redundancy,
S2 (OCL2FOL)	Class Diagrams	Yes, Subset	Unsatisfiability of OCL Constraints
S3 (FiniteSat)	Class Diagrams	No	Finite Satisfiability
S4 (AuRUS)	Class Diagrams	Yes, General	Satisfiability of the Conceptual Schema, Redundancy of Constraints Liveliness of Classes or Associations
S5 (DL)	Class Diagrams	No	Finite Satisfiability
S6 (OCL-Lite)	Class Diagrams	Yes, Subset	Schema Satisfiability, Class Satisfiability, Constraint Redundancy
S7 (OODB)	Object-Oriented DB Schemas	No	Database Constraints Satisfiability
S8 (HOL-OCL)	Class Diagrams	Yes, General	Satisfiability
S9 (UML2Alloy)	Class Diagrams	Yes, General	Weak Satisfiability, Strong Satisfiability, Liveliness of a Class
S10 (USE)	Class Diagrams	Yes, General	Satisfiability, Constraint Independence
S11 (BV-BSAT)	Class Diagrams	Yes, General	Satisfiability, Constraint independence
S12 (PVS)	Class Diagrams	Yes, Subset	N/A
S13 (KeY)	Class Diagrams	Yes, General	N/A
S14 (Object-Z)	Class Diagrams	Yes, General	N/A
S15 (UML-B)	Class Diagrams	Yes, General	Incoherent Constraints
S16 (CDOCl-HOL)	Class Diagrams	Yes, General	N/A
S17 (MathForm)	Class Diagrams	No	Consistency of a Classifier

Table 3.4: Tools, execution mode and feedback provided in each study

Study	**Tool's name**	**Feedback**	**Automation**	**Completeness**	**Termination Ensured**
S1 (UMLtoCSP)	UMLtoCSP	Yes/No + Sample model when "Yes"	Automatic	No	Yes
S2 (OCL2FOL)	Prototype	Yes/No	Yes	Solver dependent	Solver dependent
S3 (FiniteSat)	N/A	N/A	N/A	N/A	N/A
S4 (AuRUS)	AuRUS	Yes/No + Sample model when "Yes" + Hint when "No"	Automatic	Decidability analysis dependent	Only for specific OCL constraints
S5 (DL)	N/A	N/A	N/A	N/A	N/A
S6 (OCL-Lite)	N/A	N/A	N/A	Yes	Yes
S7 (OODB)	N/A	N/A	N/A	Yes	Yes
S8 (HOL-OCL)	HOL-OCL	Yes/No	Interactive	Yes	No
S9 (UML2Alloy)	UML2Alloy	Yes/No + Sample model when "Yes"	Automatic	No	Yes
S10 (USE)	USE	Yes/No + Sample model when "Yes"	Automatic	No	Yes
S11 (BV-BSAT)	Prototype	Yes/No + Sample model when "Yes"	Automatic	No	Yes
S12 (PVS)	N/A	N/A	N/A	N/A	N/A
S13 (KeY)	N/A	N/A	N/A	N/A	N/A
S14 (Object-Z)	N/A	N/A	N/A	N/A	N/A
S15 (UML-B)	Prototype	Yes/No	Interactive	Yes	No
S16 (CDOCL-HOL)	N/A	N/A	N/A	N/A	N/A
S17 (MathForm)	N/A	N/A	N/A	N/A	N/A

3.3 Challenges And Areas of Improvement

After having finished the description of the 17 studies, in this section the focus is on discussing a number of findings that, in our opinion, are worth noting.

The first important issue detected is the absence of a precise and rigorous terminology, shared among all the verification approaches analyzed. One implication of this is the difficulty for contextualizing the works in this field, especially when compared with works from other fields. In particular, the main challenge is regarding the works defining themselves as works addressing "model consistency checking". In fact, the analysis presented in this chapter has reinforced our initial impression of the existence of a gray zone when trying to determine what the exact boundaries are between model verification and model consistency checking. Some works use the term consistency checking to refer to inter-model relationships that must hold (e.g. a call to a method of a class in a UML sequence diagram should require that the same class in the UML class diagram includes the corresponding method defined). These works addressing behavioral aspects or inter-model relationships, clearly fall outside of the scope covered here. Nevertheless, other works use the term consistency checking in a broader sense overlapping with what has been defined here as satisfiability, see for instance the different notions of consistency introduced by Wahler et al. [106]. From that point of view, all the works covered here could also be regarded as consistency checking approaches.

Clearly, an unambiguous definition of all the words around the concept of model verification (including verification itself, validation, consistency, well-formedness and so forth) is needed.

The lack of a precise and rigorous terminology affects also the way correctness properties are named and defined. One example of this is the correctness property commonly referred to as "satisfiability", the most popular one among static model verification approaches. After having read the papers collected for this analysis, we realized that there were at least 6 different ways of referencing satisfiability. In some cases, this lack of homogeneity might be understandable. After all, "satisfiability" is a term widely used so it should be normal that different papers used different notions of the word (e.g. in some satisfiability may be what others call strong satisfiability while others may use the notion of weak satisfiability; with both concepts in its turn lacking also a precise and unambiguous definition). Unfortunately, the differences in meaning were not always so slightly different and sometimes different flavors of satisfiability coexisted even in the same approach. All these situations are problematic since, when the reader has a preconceived definition for the property in mind, and this is not the same as the one used in the paper, this will likely induce to errors in the interpretation of the text. Furthermore, and to make matters worse, certain correctness properties could be expressed in terms of others more "core" ones

(e.g. constraint redundancy can be expressed in terms of constraint subsumption which in turn can be expressed in terms of a satisfiability relationship).

In our opinion, this lack of homogeneity when precisely naming and defining correctness properties could be clearly improved with the creation of a catalog of correctness properties, where to find the list of correctness properties that can be checked when addressing formal verification of static models. In this catalog, precise and unambiguous names and meanings should be given to the different correctness properties, as well as a clear description of how they are related to each other. As far as we know, no efforts have been made so far in this direction.

Another finding is the difficulty to evaluate and compare the coverage and performance of existing verification tools. The majority of tools analyzed tend to be accompanied by a set of samples (a small number of input models where to check the correctness properties covered by the tool), that are usually simplistic and not representative of real scenarios. Although the existence of sample input models is always welcome, and their simplicity can be linked to the space limitations in research papers, this limits the performance analysis of the tools. And since the samples obviously vary from one tool to another, running comparisons between different verification tools is considerably more complex. Interoperability problems between modeling tools and differences on the modeling constructs each tool supports (and the terminology they use as discussed before) complicate even more the situation.

We think a possible way to improve the current situation would be the creation of a standard set of benchmarks as typically done in other communities (e.g. see the TPC transaction processing database benchmarks [12]). These benchmarks, which must be based on the catalog of correctness properties proposed before, should be composed of multiple sets of models of varying sizes and complexity, accompanied by the list of correctness properties that could be checked on them, as well as the expected results. The existence of these benchmarks could not only facilitate running performance analysis among different verification tools, but also enhance their development, since they could also be used to test the tools. As of now, we are not aware of the existence of initiatives regarding the creation of benchmarks for model verification tools.

Finally, the last and most important finding is about the adequacy of existing verification tools. Although at first sight, by looking at Table 3.4 the situation may not look so bad, the reality is that the number of verification tools available is certainly limited. If we consider that some tools were created with the intent of addressing validation issues, like USE [S10] or UML2Alloy [S9], and that tools like HOL-OCL [S8] require from the user a non-negligible expertise on formal aspects, we may conclude that the existing offer of verification tools, apart from being certainly

12. http://www.tpc.org/

limited in size, is in some cases targeted at a very limited audience.

In our opinion, a verification tool, in order to be effective and widely adopted, has to present, at least, four important characteristics: first, it should hide all the complexities derived from the utilization of formal methods, up to the point of making their presence transparent to the end user. Second, it should integrate seamlessly into the model designer tool chain. Third, it should provide a meaningful feedback. And, more importantly, four, it should be reasonably efficient, not making users to wait for ages when verifying large, real-world models. We believe these aspects are, from an end-user point of view, more important than other more formal aspects, like the completeness of the results.

Unfortunately, none of the verification tools analyzed in this study does a good job at fulfilling all these requirements. In general, these tools do not integrate well, and have been designed to conduct the verification separately from the rest of tasks that characterize the work of a model designer. When it comes to hiding the underlying formal methods employed, the situation is better, especially in the case of bounded verification approaches, although having to manually set the boundaries of the search space (as it is the case, for example, in [S1] and [S9]) can be an issue when verifying large models. The feedback can be considered acceptable when the model under analysis is found to be correct, but is clearly insufficient in the other case, with the majority of tools yielding no feedback on where to apply fixes if the model is found to be not valid (to the best of our knowledge, only [S4] provides some hints to help users on this). Finally, efficiency is a major issue. In general these tools behave well when dealing with toy examples or models of reduced size, but the performance drops dramatically when they are used to verify large models or real-world examples.

All in all, it comes as no surprise that none of these tools seem to have a strong user base. At this moment, model verification can be regarded as an unpleasant experience, that forces users to switch back and forth between model editors and verification tools to check for errors every time models are refined, usually with little or no clue on where to apply fixes if the verification fails.

We would not like to finish this section, though, without contributing some ideas on how these deficiencies could be addressed. Regarding integration with other tools and hiding complexity, a major effort on development tasks is clearly needed. Improving efficiency and feedback is clearly related to the underlying solvers employed during the verification process. These tools have experienced a dramatic improvement in the last few years, but still, even more improvements are needed. Meanwhile, doing research on techniques to cleverly define the search space boundaries of bounded verification approaches, and on incremental verification techniques, could alleviate this.

3.4 Conclusions

In this chapter the current state of the art of static model verification tools has been presented. The different approaches have been described, some areas of improvement have been identified, and some enhancement proposals have been made. In the rest of chapters of Part II we will delve more into some of these proposals. The first one, presented in the following chapter, is a static model verification tool for class diagrams and Domain Specific Modeling Languages (DSMLs) called EMFtoCSP.

4

EMFtoCSP: Static Model Verification in Eclipse

4.1 Motivation

After the analysis of the state of the art on static model verification techniques, it is clear that, unfortunately, the number of existing verification approaches is rather small, and some of them are not even supported by the presence of a tool. As an example of this, not even the Eclipse platform [1], which is perhaps the most popular software development platform at the time of promoting the utilization of model-based technologies, features a strong support for model verification activities.

Based on the Eclipse Modeling Framework (EMF) [2], which is probably de facto standard modeling framework in the industry, Eclipse offers a variety of tools and standards implementations. Some examples of this are: graphical model editors like Papyrus [3]; model transformation engines such as ATL [4]; frameworks for the development of textual DSLs, such as Xtext [5]; integrated support for UML2 and OCL [6]; frameworks for software modernization, such as Modisco [7]; tools for facilitating model weaving activities, such as the Atlas Model Weaver (AMW) [8]; and many others.

1. http://eclipse.org
2. http://projects.eclipse.org/projects/modeling.emf.emf
3. http://www.eclipse.org/papyrus/
4. http://www.eclipse.org/atl/
5. http://www.eclipse.org/modeling/tmf/?project=xtext
6. http://www.eclipse.org/modeling/mdt/?project=uml2
7. http://www.eclipse.org/MoDisco/
8. http://projects.eclipse.org/projects/modeling.gmt.amw

Unfortunately, and in spite of the amount of Eclipse projects devoted to modeling activities, there is no official Eclipse project devoted to the verification of EMF models. To the best of our knowledge, only the EMF includes some facilities to check whether the instances of a given model satisfies the OCL constraints defined at the model level. In general, this is far from enough when trying to ensure model correctness.

In order to alleviate this, in this chapter we present EMFtoCSP[9], an Eclipse integrated tool for the verification of static models.

4.2 EMFtoCSP in a Nutshell

EMFtoCSP is a tool devoted to the verification of EMF models enriched or not with general OCL constraints. The tool is an evolution of the tool UMLtoCSP [45] mentioned in Chapter 3. In EMFtoCSP, the initial model along with its constraints and the correctness properties to be checked, are translated into a CSP. Then, a constraint solver is used to determine whether a solution for the CSP exists or not. The CSP is build such that the CSP has a solution if and only if the model plus the constraints satisfy the correctness property. If a solution is found, EMFtoCSP provides a valid instance of the input model to certify it.

As of now, EMFtoCSP supports the verification of the following correctness properties: strong satisfiability, weak satisfiability, lack of constraint subsumptions and lack of constraint redundancies. It is important to notice that there is a relationship between some of these properties, for example, strong satisfiability implies weak satisfiability and the lack of constraint subsumption between two constraints implies that none of them are redundant.

Compared to UMLtoCSP, EMFtoCSP includes, among other improvements, a more general scope that can be used to verify a larger variety of models, and a revisited version of the CSP generation mechanism. In particular, the tool can also be used to analyze the quality of Domain Specific Modeling Languages (DSMLs) by evaluating the correctness of their abstract syntax (e.g. by checking if it is possible to create models conforming to that metamodel). When it comes to the generation mechanism, thanks to the work of Büttner et al. [107], EMFtoCSP supports the analysis of OCL expressions including operation on Strings in general terms.

4.3 CSP Generation

As mentioned before, the CSP generation process is essentially the same employed in UMLtoCSP [45], mostly because both tools rely on the same CSP solver,

9. http://code.google.com/a/eclipselabs.org/p/emftocsp/

the ECL^{i}PS^e Constraint Programming System [10]. In any case, in this section, this generation process is briefly described. A more comprehensive description can be found in the work of Cabot et al. [108].

For the sake of efficiency, the verification problem can be split into two different subproblems:

- **Subproblem 1**: choose a valid population size for the model, i.e. decide the number of instances of each class (objects) and association (links) that *may* provide a valid solution.
- **Subproblem 2**: given a specific population size, assign legal values to all attributes of objects and association ends of links and check if the assignment constitutes indeed a valid solution.

Both subproblems can be defined as CSPs and solved sequentially, using solutions to subproblem 1 as an input to subproblem 2. In the following, we characterize the contribution of each EMFtoCSP input element (models, constraints and properties) to each subproblem. Our running example will be the EMF model in Fig. 4.1 [109], which describes a simple metamodel for Entity-Relationship diagrams annotated with several OCL invariants.

4.3.1 Model Translation

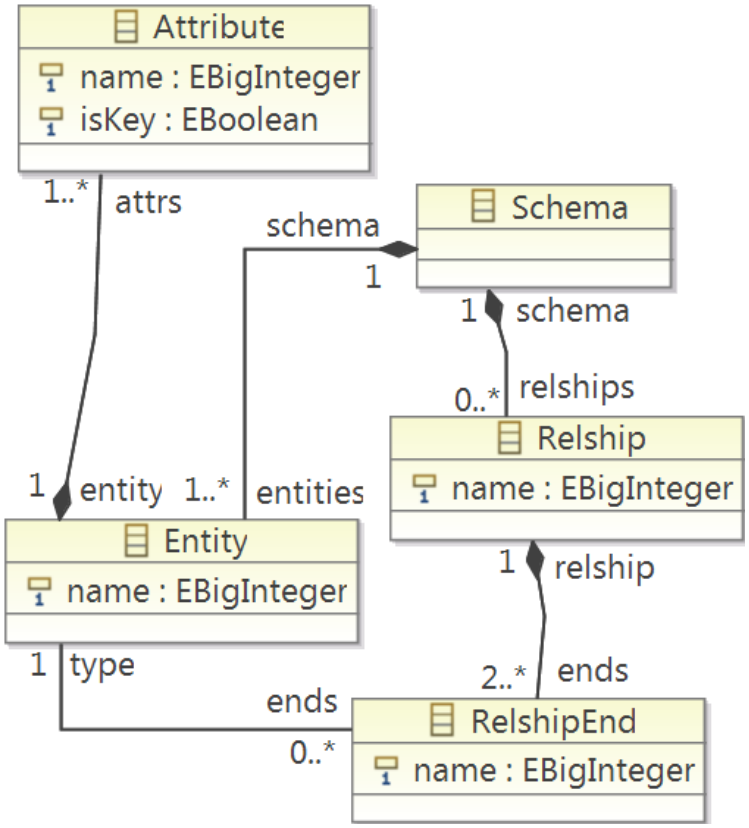
The model is the core of the CSPs for both subproblems, as it defines the relevant variables and domains:

- In subproblem 1, there is one integer variable for each class (e.g. “SizeSchema”, “SizeRelship”) and another for each association (e.g. “SizeAttributeEntity”. The domain of these variables goes from 0 to the maximum number of objects/links to be considered in the (bounded) search, a value which can be configured by the user of EMFtoCSP, beforehand.
- In subproblem 2, the number of objects and links is fixed by the previous subproblem. For each object, there is one integer variable (“oid”) and one variable per attribute (e.g. “name” and “isKey”). For each link, there is one variable per association end (e.g. “schema”, “entities”, ...). The domain of attributes is user supplied in the configuration of EMFtoCSP, while the domain of “oid” is set by the tool and directly related to the number of instances of each class. Finally, the domain of association ends is precisely the domain for “oid” in the class adjacent to the association end.

However, this is not enough, the graphical constraints in the model must be also captured in the CSPs:

- In subproblem 1, the multiplicity of association ends defines constraints over the population of the classes participating in the association. Also, inheri-

10. http://eclipseclp.org



(a)

context Schema **inv** ERN:
entities−>forAll(e: Entity | relships−>forAll(r: Relship | e.name <> r.name))

context Schema **inv** RN:
relships−>forAll(r1, r2: Relship | r1.name = r2.name implies r1 = r2)

context Schema **inv** EN:
entities−>forAll(e1, e2: Entity | e1.name = e2.name implies e1 = e2)

context Entity **inv** EAN:
attrs−>forAll(a1, a2: Attribute | a1.name = a2.name implies a1 = a2)

context Entity **inv** KEY:
attrs−>exists(a: Attribute | a.isKey = true)

context Relship **inv** REN:
ends−>forAll(e1, e2: RelshipEnd | e1.name = e2.name implies e1 = e2)

(b)

Figure 4.1: Running example: (a) Metamodel for ER diagrams, (b) OCL invariants constraining the choice of identifier names.

tance hierarchies define constraints over the population of subclasses and superclasses, e.g. each instance of a class is also an instance of its superclasses.

– In subproblem 2, the multiplicity of association ends constrains the choice of values for the association end variables: there is a lower and upper bound to the number of times that an object may participate in an association. Inheritance hierarchies constrain the assignment of "oids": an object should be given the same "oid" in a subclass as in the superclass, taking into account restrictions such as disjointness or completeness of the inheritance relationship. Finally, there are some additional well-formedness constraints that must be captured in the CSP, such as the uniqueness of "oids" within a class or the uniqueness of links in an association.

4.3.2 Constraints Translation

OCL invariants establish properties that must be satisfied by all objects of the context class. These properties are translated into constraints of subproblem 2 that refer to the variables of the CSP.

First, the OCL invariant is parsed as an Abstract Syntax Tree (AST) where each node represents an expression: intermediate nodes are the operators and method calls and leaves are constants, attribute names, ... Each expression is translated into an ECL^{i}PS^{e} predicate "eval(Instances, Result)" that receives all variables of the CSP as a parameter and characterizes its result:

– **Leaf nodes** either set a constant value for the result or relate it to the value of a variable of the CSP. For example, the boolean constant `false` is translated into the predicate:

```
evalConstantFalse( _, Result ) :- Result = 0.
```

where "_" states that the result of this predicate does not depend on the rest of the variables of the CSP.

– **Intermediate nodes** describe the result as a combination of the result of its subexpressions. For instance, a node with a boolean implication operator would be translated into the following predicate:

```
evalImplies( Instances, Result ) :-
  eval1stChild( Instances, Result1 ),
  eval2ndChild( Instances, Result2 ),
  =>(Result1, Result2, Result).
```

This predicate does not compute the implication, as the variables of the CSP

("Instances") do not have a value until a solution to the CSP is found. Instead, it states the relationship between the result of the implication and its subexpressions. This relationship will be used to guide the search process for a feasible solution, e.g. if "Result1" is false, then we know that "Result" is true without having to evaluate "Result2".

Finally, the ECL^{i}PS^e predicate for the root node of the AST is evaluated on all objects of the context class, forcing its result to be true, that is:

```
evalRoot(Instances, 1).
```

4.3.3 Properties Translation

As mentioned in the previous chapter, correctness properties state desirable conditions about models in order to be considered correct. Given a model M and a correctness property P, the goal is to compute a legal instantiation of M (one that satisfies all graphical and textual constraints of M) that is a "witness" of P, i.e. it proves that M satisfies P.

It was also mentioned before that these properties could be classified in two different families: conditions about the "population" of the model, e.g. that it is not empty, or about the "relationship among constraints", e.g. that no pair of invariants is equivalent. These conditions are encoded in the CSP as additional constraints in subproblem 1 (for conditions on the population size) or subproblem 2 (for conditions about the relationship among invariants).

As of now, EMFtoCSP is capable of verifying the following correctness properties:

- **Strong Satisfiability**: It is possible to find a legal instance of the model with a non-empty population for all classes and associations.
- **Weak Satisfiability**: It is possible to find a legal instance of the model with a non-empty population for some class.
- **Lack of Constraint Subsumptions**: Given a model with two OCL constraints C_1 and C_2, it is possible to find a finite instantiation where C_1 is satisfied and C_2 is not.
- **Lack of Constraint Redundancies**: Given a model with two OCL constraints C_1 and C_2, it is possible to find a finite instantiation where only one constraint is satisfied. Constraints C_1 and C_2 are called redundant otherwise (both have always the same truth value).

The translation of these properties into ECL^{i}PS^e constraints is straightforward. For example, weak satisfiability requires that the sum of all size variables in subproblem 1 is greater than zero:

```
weakSatisfiability( SizeVars ) :- sum(SizeVars) #> 0.
```

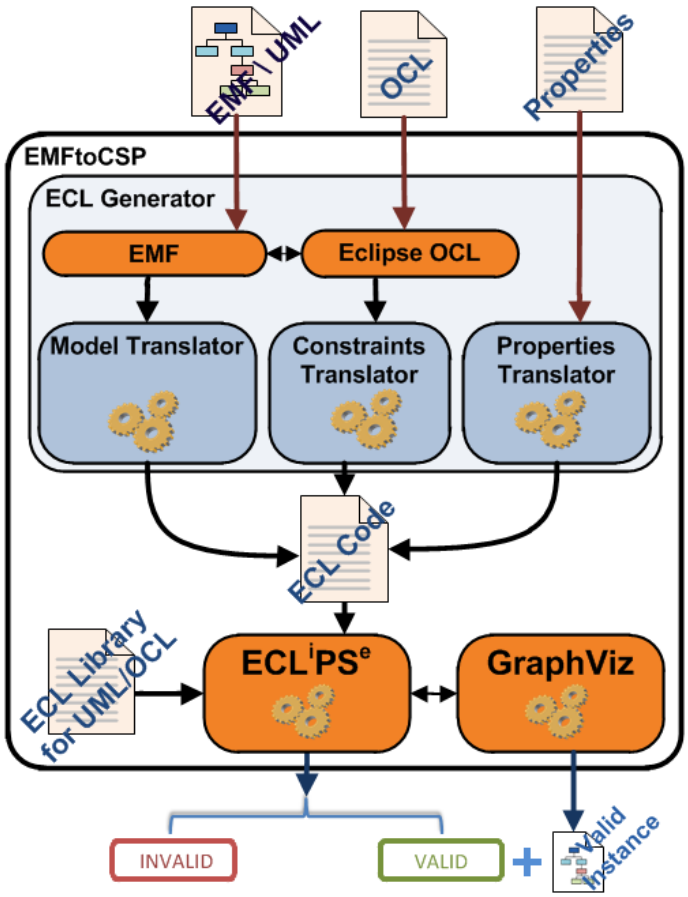


Figure 4.2: EMFtoCSP architecture

As another example, to check if constraints RN and EN from Fig. 4.1(b) are non-redundant, a constraint is added to subproblem 2 stating that the root predicates of RN and EN evaluate to a different result:

```
nonRedundant_RN_EN( Instances ) :-
  evalRootRN( Instances, Result1 ),
  evalRootEN( Instances, Result2 ),
  Result1 #\= Result2.
```

4.4 The Tool

Once the generation of the CSP has been introduced, we now describe the EMFtoCSP tool itself.

4.4.1 Architecture

The tool architecture can be seen in Fig. 4.2. User inputs are managed by the subsystem called “ECL Generator”, which is in charge of generating the code

to feed the CSP solver with. In this subsystem, three different components are clearly distinguished each one coping with the different input elements that need to be translated, namely, the model, the set of constraints over the model and the properties to be checked. The EMF and Eclipse OCL [11] parsers are used in the process.

Once the translation process has been performed, the generated CSP (depicted as "ECL code" in the figure) is sent to ECL^{i}PS^e to check whether the input model holds the properties selected. Once ECL^{i}PS^e finishes the search of a solution for the CSP, its feedback is interpreted and a message informing whether the input model holds the selected properties or not is displayed to the user. If the result is positive, the tool GraphViz is used to graphically display the valid instance of the model identified as a solution by the solver.

It is worth noting that EMFtoCSP has been designed keeping in mind several possible extensions in the future. For instance, it is possible to plug modules translating models into other formalisms than the one used by ECL^{i}PS^e, as well as other solvers, provided that these modules respect the defined interfaces.

4.4.2 Usage

EMFtoCSP is, in essence, a collection of Eclipse plug-ins, so, in order to use the tool, these plug-ins must be loaded into the Eclipse platform. After doing this, and having the Eclipse platform up and running, the tool can be launched by right-clicking on a model on the "Package Explorer" view and choosing "Validate model..." from the context menu [12]. This will display the EMFtoCSP GUI.

As can be seen in Fig. 4.3, EMFtoCSP provides a GUI in the form of an easy-to-use wizard that guides the user through a sequence of predefined steps to collect the user input for the verification process. The first step, shown in Fig. 4.3(a) is optional and consists in selecting the file with the OCL constraints of the model. This step is optional because of two reasons: first, not every model has to be enriched with OCL constraints, and second, EMFtoCSP also supports EMF models with embedded OCL constraints. In this last case, a separate text file with OCL constraints is not usually needed.

The second step is setting the search space boundaries. These boundaries, as it can be seen in Fig. 4.3(b), have to do with the specification of maximum and minimum limits for the number of class and association instances that valid model instances can have, and with setting the domain values for the different class attributes in the model.

11. http://www.eclipse.org/modeling/mdt/?project=ocl
12. The launcher of EMFtoCSP can only be accessed from the "Package Explorer" view, so it is important to choose a perspective in which this view remains visible

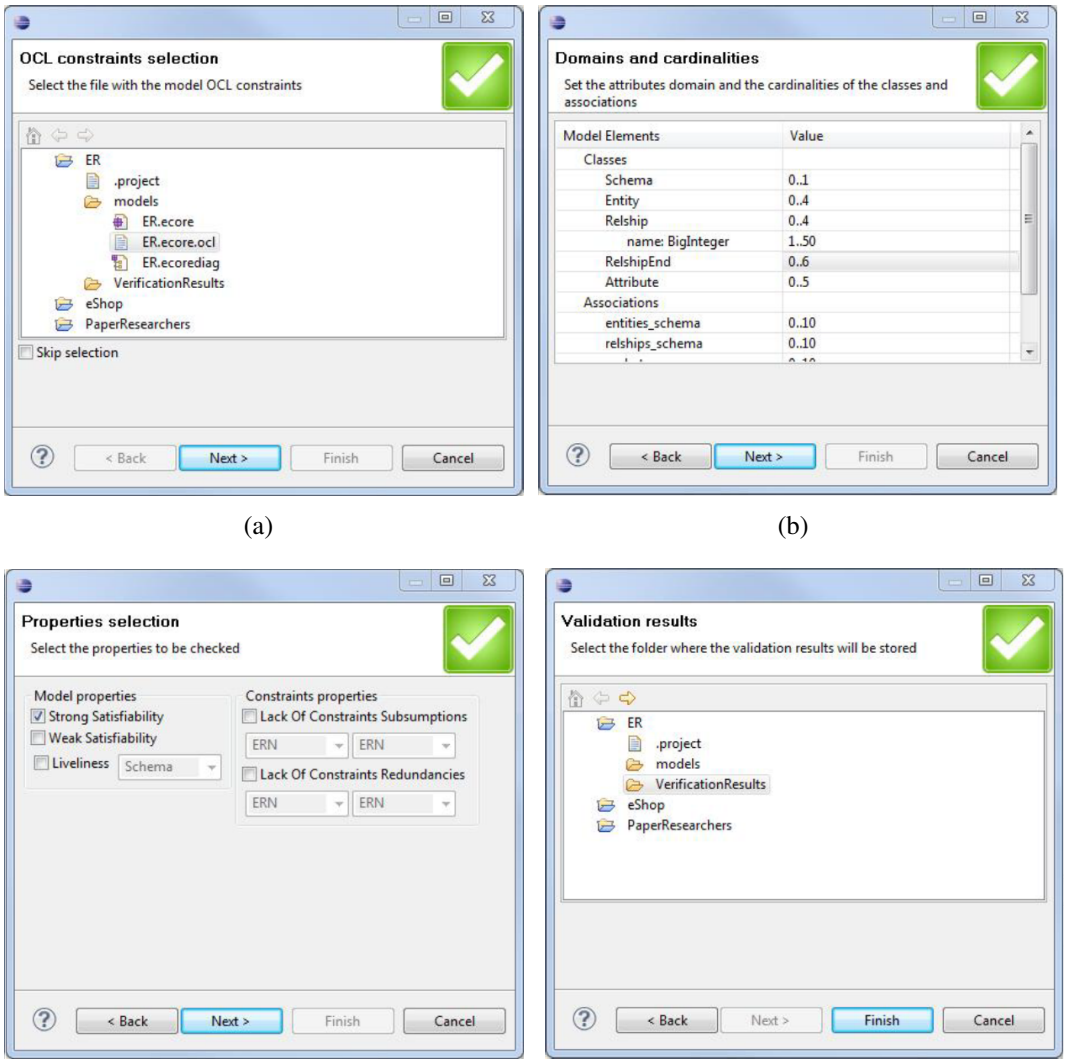


Figure 4.3: EMFtoCSP Graphical User Interface.

In the third step, the election of the correctness properties to be checked is made. Fig. 4.3(c) shows the list of correctness properties available. Check boxes are used to adjust the selection.

Finally, the last step (Fig. 4.3(d)) implies establishing a location where to store the outputs of the verification process. This is important because, no matter whether the model is found to be correct or not, EMFtoCSP always provides the source code of the CSP generated as input for the CSP solver.

Once EMFtoCSP has been configured, the verification process can be executed by clicking "Finish" on the wizard. As a result, EMFtoCSP will display a message informing the user whether the input model satisfies the selected properties or not. If it does, EMFtoCSP will additionally output a valid instance of the input model that proves the property. In order to do this, EMFtoCSP, as it was the case of UMLtoCSP, requires the presence of the graph visualization package Graphviz [13].

13. http://www.graphviz.org/

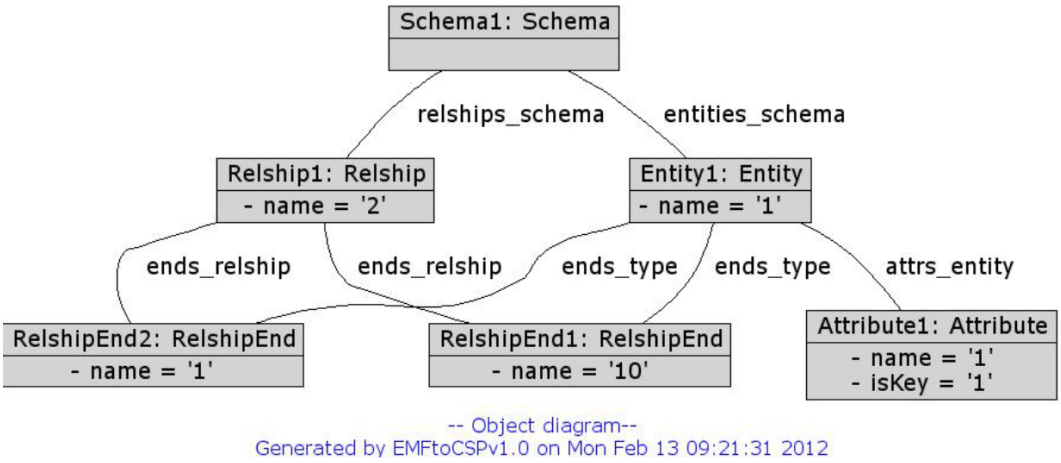


Figure 4.4: Valid instance of the running example model

An example of this can be seen in Fig. 4.4, where a valid instance of the metamodel of Fig. 4.1 is displayed as a solution for the inputs provided in Figs. 4.3(a), 4.3(b) and 4.3(c).

4.5 Performance

A tool like EMFtoCSP is only useful as long as it can scale when applied beyond toy examples. Regarding this, we have performed some experiments that show the applicability of the tool. The fact that the designer can decide herself the limits of the search space also facilitates using the tool with large models, where she can start by verifying the model using a small search space and expanding it later on if necessary.

In general, the scalability of the tool depends on the constraints of the model and the generated CSP. Two main factors are: (1) how much of the CSP can be solved using constraint propagation (and therefore avoiding backtracking), and (2) whether the CSP is non-trivially unsatisfiable (for the reason of symmetries).

This can be illustrated using the ER example from Fig. 4.1. Table 4.1 shows the runtimes of EMFtoCSP for several satisfiable ranges. All tests were conducted on a standard 2.2Ghz office laptop running Windows 7 and ECL^{i}PS^{e} 6.0 with default settings. The ranges for "RelshipEnd", the name attributes, and all link set sizes were set to 0..1000. We can see the tool finds ER instances of up to several hundred objects (in total) in reasonable time. For these cases, EMFtoCSP efficiently finds a valid link set (using linear constraint propagation to determine a valid link set size in the first step and using a global cardinality constraint as described in [110] in the second step). For this link set, a valid assignment of all attributes is then determined using linear constraint propagation, because at this time, the universal quantifiers have been completely unrolled, leaving a purely linear problem. To

Table 4.1: Runtimes for SAT cases of ER

Entities	Relships	Attributes	Runtime
1	1	1	$\leq$ 0.1s
10	10	50	0.76s
10	20	50	1.45s
20	20	50	1.99s
50	10	50	2.13s
50	20	50	3.30s
20	20	100	5.52s
50	20	100	9.71s
20	50	100	17.89s
50	50	100	24.91s

make sure solutions found are non-trivial (e.g., all attributes connected to the first entity), we verified that the resulting runtimes are similar when changing the upper multiplicity bounds of the roles "entities", "relships", and "attributes" from "*" to 0..10.

Table 4.2 shows the runtimes for several "hard" unsatisfiable cases. In this setting, we restricted the range of the name attributes (i.e., the number of different names per type), so that there are not enough names to fulfill the corresponding constraints of the model. The table shows that EMFtoCSP scales much worse for these cases. The reason is that the (failing) attribute assignment is tried for all symmetrical link sets before reporting UNSAT. We hope to partly address this issue using a symmetry breaking during search approach such as described in [111].

While the employed example is very small in terms of the number of classes, we want to stress that our tool can also solve larger class diagrams, as the complexity of the search problem is not directly related to the number of classes in the model. On the contrary, given a number of objects (such as the 200 objects in the last line in Table 4.1), the search space is typically even smaller when these objects are distributed on more classes, because there are less possible combinations.

Table 4.2: Runtimes for UNSAT cases of ER

Entities	Relships	Attributes	Names	Runtime
2	2	2	1	$\leq$ 0.1s
3	3	3	2	$\leq$ 0.1s
4	4	4	3	$\leq$ 0.1s
5	5	5	4	$\leq$ 0.1s
6	6	6	5	0.43s
7	7	7	6	5.77s
8	8	8	7	93.08s

4.6 Conclusions

In this chapter we have presented EMFtoCSP, a tool for the fully automatic, decidable and expressive verification of EMF models extended with OCL constraints, based on their translation into a CSP such that the CSP has a solution if and only if the model satisfies the chosen correctness property.

EMFtoCSP, like its predecessor UMLtoCSP, follows a bounded verification strategy that ensures termination by limiting the search space when looking for a solution for the CSP. Limits are created by restricting the number of instances per class and association and the domains of each attribute in the model. The trade-off is that the verification process is not complete (i.e. the CSP may have a solution beyond the considered search space).

EMFtoCSP is, to the best of our knowledge, the first tool for the verification of static models in the Eclipse platform.

5

Improving Static Model Verification Performance (I). Incremental Verification of Models

5.1 Motivation

After having analyzed the current landscape of approaches devoted to the verification of static models, it is clear that, the most common way of addressing this problem is by translating the model, along with the correctness properties to be checked, into some kind of formalism, which is then exploited during the reasoning process, to determine whether the model satisfies the properties under scrutiny. This reasoning process is conducted with the help of tools like SAT or CSP solvers, specialized in reasoning over the chosen formalism. However, due to the nature of the problem, these tools tend to experience performance issues that thwart their utilization, specially when dealing with large, real-life models. If we add to this, that models are not immutable (i.e. they are subjected to modifications to better reflect the reality of the systems under analysis, thus making advisable to verify them over and over again), it stands to reason the low rate of adoption of this kind of tools.

A necessary first step to increase the rate of adoption of these tools is therefore to enhance the performance of the verification process itself. In this regard, a possible alternative to achieve this could be conducting the verification in an incremental fashion where every time the model is modified, only the relevant parts are verified again. This way, the size of the model that really needs to be verified could be reduced, which might help to improve the performance of the verification process.

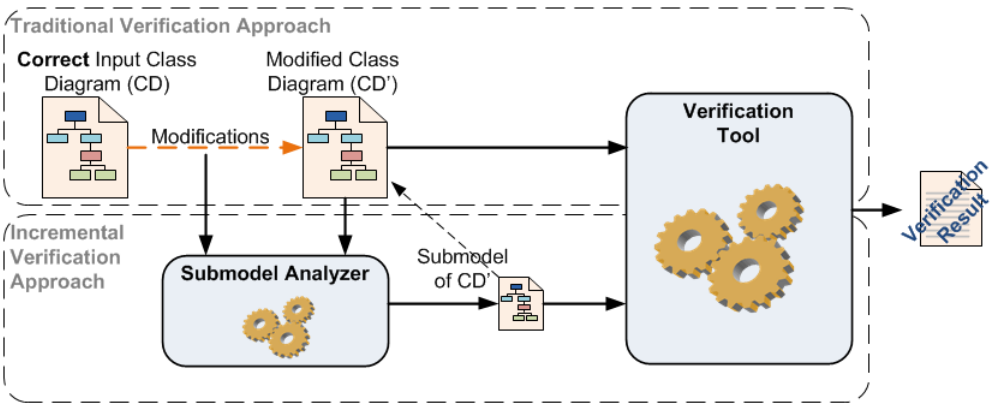


Figure 5.1: Traditional Verification vs Incremental Verification

In this chapter we present an incremental verification approach for checking correctness in UML/OCL class diagrams [1]. This approach automatically calculates the submodel that needs to be verified, every time the model is modified (a prerequisite of our method is that it is assumed that the model was correct before modifications were made). This way, there is no need to verify the whole model every time. Besides, the mechanism proposed here can be easily integrated with model editors, thus facilitating its utilization by model designers. Finally, and because of the submodel calculation, this method provides a more meaningful feedback when the model under analysis is found to be not correct.

5.2 The Method in a Nutshell

Fig. 5.1 illustrates the differences between traditional verification approaches and our proposal for incremental verification. Traditional approaches verify the whole model every time, regardless of the nature of the modifications performed. The incremental approach also relies on the verification tools used by traditional approaches, but instead of verifying the whole model, it analyzes the modifications made on the model, discards those ones that have no impact on its correctness, and uses the rest to calculate the part of the model that really needs to be verified again.

It is important to state though, that an incremental verification approach is highly-dependent on the correctness property that is to be checked. Our approach focuses on satisfiability; first, because it is the most fundamental correctness property to date, and second, because other correctness properties can also be expressed in terms of this fundamental property.

Once the approach has been outlined, the following sections enter into detail on its different stages, namely, checking which modifications may impact model

1. due to the existing similarities between UML/OCL class diagrams and EMF models, this approach can also be applied to the latter without loss of generality.

correctness (thus requiring the model to be verified) and if so, what parts of the model need to be rechecked.

5.3 Modifications That May Impact Model Correctness

Given an UML/OCL class diagram, any modification over it can be analyzed from the point of view of how it affects the constraints defined in the model (by means of graphical elements or OCL constraints). If a certain modification clearly weakens those constraints, or add new constraints that clearly do not contradict any of the existing ones, then it can be assumed that there is no impact on model correctness and therefore, the modified model does not have to be verified. If, on the contrary, modifications on the model add new constraints that may contradict any of the existing ones, or produce a completely different set of constraints that is not clearly equivalent to the original one, then it is assumed that model satisfiability might have been compromised and therefore, the modified model has to be verified.

Regarding this last case, it is important to follow a conservative approach. If after one modification is made, it is complex or very time-consuming to determine how restrictive the new set of constraints is compared to the old one, then it is better to proceed with the verification. After all, existing verification tools rely on tools like SAT or CSP solvers, that can handle these scenarios more efficiently, than manually trying to find out how an intricate set of constraints relate to each other.

In the following subsections, we analyze which of the modifications most typically made over UML/OCL class diagrams may impact their satisfiability. In particular, the modifications analyzed are: adding or removing a class, modifying a class (changes on the class name, list of attributes or list of operations), setting a class as abstract or non-abstract, adding or removing a binary association, adding or removing a generalization, and adding or removing an OCL invariant. We consider the modification of binary associations, generalizations or OCL invariants as a combination of the operations “remove” and “add” over these elements.

At this point is important to make two important remarks, though. The first one is that some of these modifications when conducted in isolation can lead to consistency issues (e.g. deleting a class which is constrained by an OCL invariant). In some cases, modeling tools do a good job to avoid these inconsistencies (e.g. when a class linked to another class by means of a binary association is removed, the tool normally deletes the association too), but this is not always the case. For this reason, our analysis was conducted regardless of what the behavior of a particular modeling tool might be. An immediate consequence of this is that, as it was also the case for traditional verification approaches, incremental verification approaches

should be complemented by the presence of mechanisms to ensure model consistency. The second remark is that modifications covered by our analysis are treated as if they were atomic operations (again, without regard to what the behavior of modeling tools might be). An example of this, is the addition of a binary association; we assume that setting the cardinalities of the association ends is part of the "add" operation, and not a separated one.

5.3.1 Modification Over Classes

We start our analysis by studying the modifications on classes. Regarding this, adding classes to a model or deleting classes from a model does not impact its satisfiability. Neither adding classes nor deleting them strengthens the original set of constraints over the model. Regarding class modification, the same goes for changes on the class name or on the list of attributes or methods. However, changes on the "isAbstract" modifier may impact the satisfiability of the model, specially when the class is part of associations or generalizations. In this last case, our method does not calculate the submodel affected and the model in its entirety must be verified.

5.3.2 Modification Over Binary Associations

The second group of modifications studied are the ones that involve binary associations. In this analysis, binary associations are of special importance because the cardinalities at their ends may restrict the population of the linked classes. Our analysis focuses on creating and deleting associations, since modifying associations can be seen as a combination of these two operations. Here, the easiest scenario is deleting an association. This modification either weakens or does not affect the set of constraints over the two linked classes, therefore it does not impact the satisfiability of the model. Adding associations leads to more complex scenarios, although it is possible to identify certain cases when this operation has no effect on the satisfiability of the model. Fig. 5.2 shows an association with generic cardinalities M and N, and below two specific examples of valid instances when cardinalities are restricted to a certain group of values. The figure seeks to illustrate that instantiating an association with cardinalities like the ones in the two examples does not impose any stricter restriction on the minimum number of instances of the linked classes that can be created. This is because the minimum cardinality for the classes in the examples are 0 or 1, and instantiating at least one instance of every class in the model is already enforced by checking strong satisfiability. Therefore, associations with cardinalities that are combinations of the ones shown in the two examples does not impact the satisfiability of the model. For the rest of scenarios related to the creation of associations, we assume the opposite.

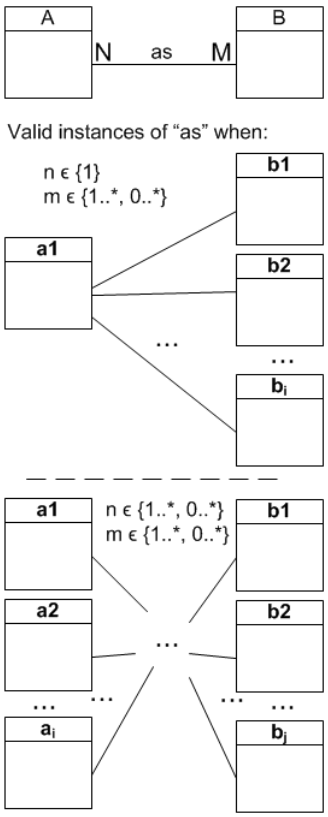


Figure 5.2: Association Cardinalities and Strong Satisfiability

5.3.3 Modification Over Generalizations

The third group of modifications studied affect to generalizations. In this case, two types of modifications are considered in our approach: creating and deleting generalizations (modifying a generalization, as it was the case for associations, can be seen as a combination of deletion and creation). Analyzing the restrictions derived from these two operations can be tricky, because of the number of complex scenarios that can appear depending on the degree of interconnections between the classes affected by the generalization and the rest of elements of the model. For this reason, we assume that creating and deleting generalizations may impact the satisfiability of the model.

5.3.4 Modification Over OCL Constraints

Finally, the modifications that can alter the set of OCL invariants of the model. The easiest scenario here is the deletion of OCL invariants. This modification weakens the set of constraints over the model and therefore, does not impact its satisfiability. However, when it comes to the addition of new invariants or the modification of existing ones, the situation differs. Due to the flexibility of the OCL, these types of modifications can lead to a plethora of complex scenarios, thus making hard and time-consuming trying to determine in a systematic way in which cases the satisfia-

bility of the model may be compromised. Because of this, we assume that creating or modifying OCL invariants may impact the satisfiability of the model.

5.3.5 Ignoring Modifications Impacting Model Correctness

Once it has been analyzed how model modifications can impact model correctness, it is necessary to consider that, in certain scenarios, not all of the modifications impacting model satisfiability, need to be taken into account at the time of identifying the part of the model that has to be reverified.

There are at least two reasons that can justify this. The first one is that some groups of modifications are of opposite nature and when applied on the same model element their effects are thwarted (for example, adding an OCL constraint and deleting it right afterward). The second one is that the same type of modification can be successively repeated on the same model element (for example, a succession of modifications on the multiplicities of a given association end). For these reasons, the last step when analyzing the list of modifications is to calculate the "net modification" on each model element. This can be done by traversing the list of relevant modifications grouped by model element, looking for these patterns and simplifying them accordingly when found.

5.4 Submodel Construction

After the analysis of model modifications that may impact model correctness (strong satisfiability, in our case), it is the time of calculating the part of the modified model that needs to be reverified.

Table 5.1 shows the pseudocode of the algorithm that calculates this. The algorithm requires two inputs: the UML/OCL class diagram resulting of the modifications ($CD : ClassDiagram$), where $ClassDiagram = \langle C, A, G, I \rangle$ is a collection of Classes (C), Associations (A), Generalizations (G) and OCL invariants (I), and the list of modifications ($ML : Modifications$). The output is the submodel ($SM : ClassDiagram$) of the input class diagram CD that must be fed to the verification tool.

Listing 5.1: Algorithm to Calculate the Submodel

```
INPUT:
  CD: ClassDiagram
  MODLST: ModificationList
OUTPUT:
  SM: ClassDiagram
```

```
ForEach (Modification M in MODLST)
  AMELST = AMELST + M.GetAffectedModelElements()
SM = CreateEmptyModel()
while (AMELST is NOT EMPTY)
  ME = ExtractFirstModelElement(AMELST)
  SM = SM + ME
  if (ME in CD.GetAssociations())
    AMELST = AMELST + GetAssociationLinkedClasses(ME)
  endif
  if (ME in CD.GetClasses())
    AMELST = AMELST + GetLinkedNonZeroOneAssociations(ME)
    AMELST = AMELST + GetLinkedGeneralizations(ME)
    AMELST = AMELST + GetOCLInvariants(ME)
  endif
  if (ME in CD.GetGeneralizations())
    AMELST = AMELST + GetGeneralizationLinkedClasses(ME)
  endif
  if (ME in CD.GetOCLInvariants())
    AMELST = AMELST + GetOCLLinkedClasses(ME)
    AMELST = AMELST + GetOCLNavigatedAssociations(ME)
  endif
endwhile
```

The algorithm starts by extracting the list of model elements affected by the modifications. Then, these elements are traversed. With each iteration, the current element is incorporated to the submodel and the model elements "connected" to it are added to the list of affected elements. These model elements are considered as transitively affected by a modification and therefore must be processed. The algorithm stops when there are no more elements to visit in the list of affected model elements. We assume here that the introduction of new elements in the list of affected elements is done in such a way that model elements are not visited more than once. Finally is worth noting the difference between the methods "GetLinkedNonZeroOneAssociations()" and "GetOCLNavigatedAssociations()". The first one collects, among the associations the class is linked to, those with other cardinalities than the ones in the associations of Fig. 5.2. The same reason given in the discussion about cardinalities in the previous section applies here. The second method, "GetOCLNavigatedAssociations()", collects all the associations navigated by an OCL invariant regardless of their cardinalities. This is because when an association is navigated by an OCL invariant that is part of the submodel, then the association must be part of the submodel too, regardless of what its cardinalities are.

5.5 Experimental Results

In this section, we try to find out whether the utilization of an incremental verification approach can speed up the verification of class diagrams. To this end, we implemented a prototype of the approach described here, with the intent of running an experiment.

The experiment consists in verifying three class diagrams using EMFtoCSP and then, making three series of modifications over each one, repeating the verification after each round of changes. Every time one of the class diagrams is verified, we measure its size and how long the verification takes. Once the process is completed and each class diagram has been verified four times, we start over but this time, following an incremental verification approach with the help of the prototype tool. In order to reduce bias when doing the measurements, this tool will also rely on EMFtoCSP as the underlying verification tool.

The first step is creating the models for the test. In relation to this, we automatically generated three class diagrams of different size: 15, 45 and 90 model elements. In the rest of this section, we refer to them as small model, medium size model and large model, respectively.

The second step is to determine what modifications are made on the models. Regarding this, we designed three groups of modifications for each class diagram: M1, M2 and M3. M1 groups the modifications that strengthen the original set of constraints over the model. M2 groups the modifications that do the opposite, that is, they weaken those constraints. Finally, M3 combines modifications of the two types.

Tables 5.1, 5.2 and 5.3 show the results obtained after running this experiment on a computer equipped with a Intel Core i7 M 640 2.8 Ghz processor and 4Gb of RAM. Time was measured in milliseconds. The numbers between parenthesis in the column headers indicate the number of modifications made on each case. In the rows corresponding to the traditional verification approach, the size of the verified model is also shown between parenthesis. Same convention is followed in the rows corresponding to the incremental approach, to show the size of the calculated submodel and the time needed to calculate it. It is important to mention that there are some cases in which it was not possible to take measurements. This is because we set a time-out limit of 15 minutes for the verification process.

According to the information in the tables, the incremental approach performs slightly better in every case. However, it seems that there is no direct relation between the size of the model and the time needed to verify it. In some cases, verifying a submodel that is roughly half the size of the whole model, takes almost the same time than verifying the whole model. However, since the time needed to calculate the submodel is negligible compared to the time spent during the verification,

Table 5.1: Comparison of approaches when verifying a small model

Verification Approach	**Original Model**	**Model After M1 (4)**	**Model After M2 (5)**	**Model After M3 (2)**
Traditional	1665 ms (size: 12)	11226 ms (size: 15)	1685 ms (size: 12)	1765 ms (size: 11)
Incremental	0 ms (size: 0) (0 ms)	1830 ms (size: 12) (10 ms)	1570 ms (size: 5) (5 ms)	1656 ms (size: 6) (26 ms)

Table 5.2: Comparison of approaches when verifying a medium size model

Verification Approach	**Original Model**	**Model After M1 (12)**	**Model After M2 (15)**	**Model After M3 (4)**
Traditional	1896 ms (size: 36)	- (size: 45)	3386 ms (size: 36)	4103 ms (size: 33)
Incremental	0 ms (size: 0) (0 ms)	- (size: 36) (110 ms)	3057 ms (size: 15) (63 ms)	3229 ms (size: 18) (86 ms)

Table 5.3: Comparison of approaches when verifying a large model

Verification Approach	**Original Model**	**Model After M1 (24)**	**Model After M2 (30)**	**Model After M3 (8)**
Traditional	6896 ms (size: 72)	- (size: 90)	6817 ms (size: 72)	6318 ms (size: 66)
Incremental	0 ms (size: 0) (0 ms)	- (size: 72) (66 ms)	3923 ms (size: 30) (74 ms)	4231 ms (size: 37) (64 ms)

calculating this submodel seems to be a good decision.

5.6 Conclusions

This chapter presents an approach for the automatic incremental verification of class diagrams. Given a class diagram and a series of modifications made on it, the approach works by analyzing these modifications to identify a subset of the modified class diagram such, that the verification of this subset yields the same results than verifying the whole class diagram. A benefit of this approach is that is independent of the underlying formalism employed to verify the model, and therefore can be combined with any of the existing verification tools.

A prototype of the approach has been implemented on top of the EMFtoCSP tool. Experimental results seem to indicate that compared to traditional verification

tools, incremental verification speeds up the verification process. However, and in relation to this, further experimentation is required, especially in real world scenarios, since the size of the calculated submodel does not seem to be indicative of the speedup obtained.

6

Improving Static Model Verification Performance (II). Tightening Search Space Boundaries

6.1 Motivation

One of the most popular strategies when verifying static models is to follow a bounded verification approach consisting in limiting the search space where to look for a solution of the problem. Although the main drawback of this strategy is well known (conclusions about model correctness cannot be drawn if no solution is found within the considered search space), in the last few years, more and more approaches are adopting this strategy [45, 73, 81, 82]. The reason for this is that a bounded verification approach allows the verification process to be automatic, without restricting the expressiveness of the modeling language, or requiring users to be aware of the mathematical formalism employed in the process.

Unfortunately, setting the search space boundaries has proven itself to be a major limiting factor, since existing tools provide little support on this, either by setting inadequate default values; or by forcing users to manually define these boundaries. Manual setting of these boundaries is a big issue since it requires the specification of upper and lower bounds for the population of each class and association in the model, as well as, finite domains for each attribute. When facing big models, this translates into setting boundary values for a set of modeling elements ranging in the hundreds, or even more. On top of that, there is no guarantee that this process does not have to be repeated several times, in order to fine-tune the resulting search

space. And, of course, there is always the risk the designer makes a mistake when defining the domains, for example missing an implicit constraint or interaction between constraints, leading her to believe the model is unsatisfiable when it is just a mistake in the selection of bounds.

The reason why existing approaches do not offer better support on this task is because choosing optimal bounds automatically is as complex as the verification problem itself, so the use of heuristics or approximate methods is required. For example, the "small scope hypothesis" [97] claims that a large amount of faults can be detected by inspecting a small domain. Hence, many tools advocate for an "incremental scoping" strategy: start with small domains to get feedback quickly and progressively increase the domain size in later executions until a fault is detected or a sufficient level of confidence in the result is achieved. However, beyond that, users must select domains on their own.

In addition to increasing the usability of bounded verification approaches, bound reduction techniques can also be useful at the time of increasing their performance, since reducing the size of the search space to be explored accelerates the verification process. In this regard, and as it is shown later, tightening bounds does not add a lot of overhead to the whole verification process, and preserves any of the valid instances serving as a solution for the problem that would exist within the original bounds.

In this chapter, we present a technique developed in collaboration with Robert Clarisó [1]that can assist users of bounded verification tools to effectively set the boundaries of the search space. This approach starts from a set of initial bounds (which may be infinite) and takes advantage of all implicit and explicit constraints in the model to tighten those bounds as much as possible. To this end, an efficient technique called interval constraint propagation, which does not require solving the verification problem, is used to discard unproductive values from domain bounds.

6.2 Background on Bound Reduction Techniques

Bound reduction is a well-studied problem in the field of static program analysis. Techniques such as "interval analysis" or "shape analysis" can infer bound information about the program variables and data from the program dataflow and call graphs. This information can later be used in the bounded verification of the dynamic behavior of programs. In relation to this, different tools have been proposed for the static analysis of code written in popular programming languages like C [112–114], Java [115–117] or even Ruby [118]. Among them, we highlight the most related tool in terms of bound reduction, TACO [117], a tool for the verifi-

1. Robert Clarisó is a member of the IT, Multimedia and Telecommunication Department at Universitat Oberta de Catalunya (UOC) in Spain

cation of JML-annotated Java programs. TACO attempts to eliminate individual values from domains by calling the solver with a specially tailored formula for each value before analyzing the entire program. This allows a more fine-grained bound refinement than using intervals but, on the other hand, a time threshold must be set to avoid wasting too much time on each call.

To the best of our knowledge, only the work of Yu et al. [119] has addressed bound reduction techniques on the field of model verification, although it includes dynamic behavior. The approach considers the analysis of the size of collections in OCL invariants, pre- and post-conditions. However, the constraints generated by their analysis are only used to check size-specific properties of collections (for example retrieving an element from an empty collection).

6.3 Tightening Search Space Boundaries in a Nutshell

As shown in Fig. 6.1(b), we propose an alternative approach that optimizes the set of bounds (either from scratch or by refining an initial set of bounds proposed by the designer). These bounds can then be passed to a bounded verification tool together with the translation of the initial model (Fig. 6.1(a)). The computation of the bounds works by collecting all implicit and explicit constraints from the UML/OCL model, and expressing them as a CSP on a set of variables representing the bounds we should use when verifying the model. These bounds are then calculated by the solver relying on interval constraint propagation techniques. This process is not optimal but it is safe, that is, it may fail to compute the tightest bounds, but, as mentioned earlier, it will preserve any of the valid instances serving as a solution for the problem that would exist within the original bounds.

Fig. 6.2 shows an UML/OCL model describing the relationship between machines and their parts. This model will be used as an example throughout the chapter. By analyzing association end multiplicities, inheritance hierarchies and OCL invariants, it is possible to find constraints that limit the number of classes and associations that any valid instance must have, as well as the size of the domain of the class attributes. For example, association end multiplicities indicate that there are four parts for each machine; the OCL invariant “MachineAvailability” states that there must be at least two machines available at all times, one “cutter” and one “grinder”; and the OCL invariant “UniqueSerial” implicitly establishes that the population of class “Part” cannot be larger than the number of values in the domain of its attribute “serial”.

All these constraints can be used to automatically infer bounds without any input from the user. For example, it is possible to infer a lower bound of 1 for classes

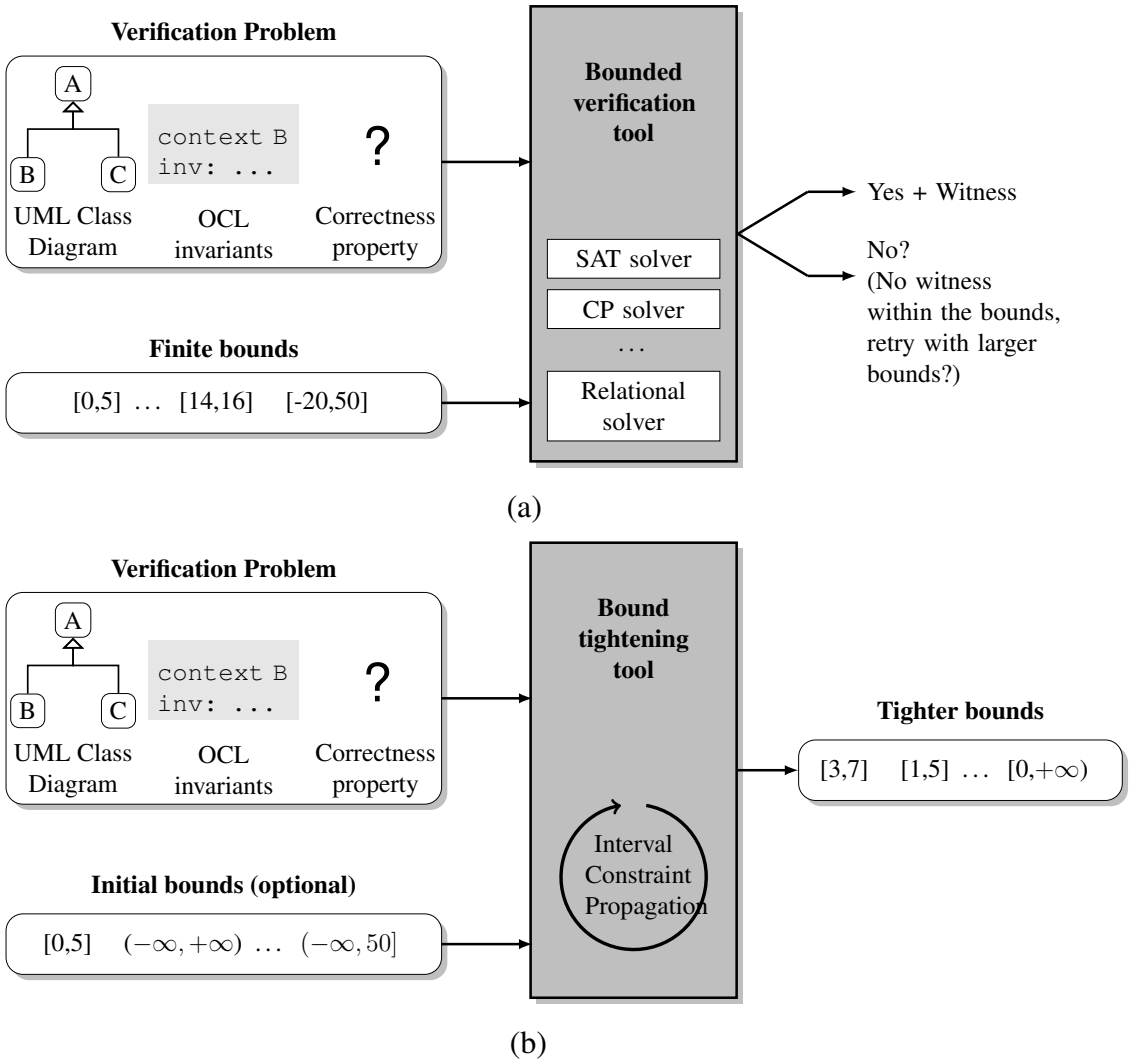
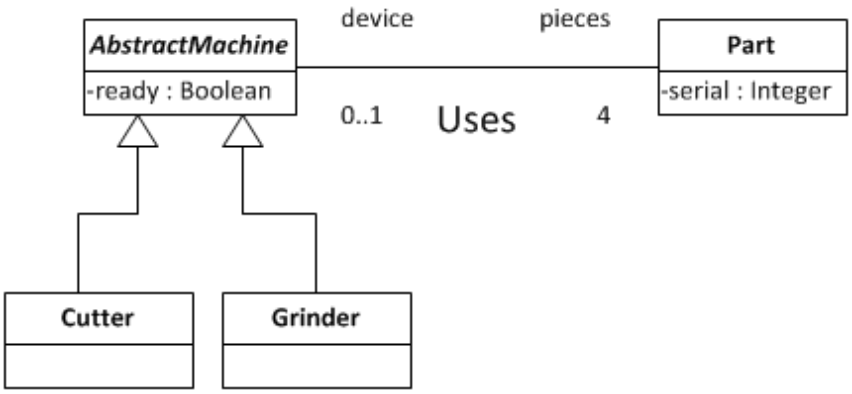


Figure 6.1: Comparison of verification approaches: (a) Typical flow with a bounded verification tool, (b) Approach proposed.

"Cutter" and "Grinder", a lower bound of 8 for class "Part", and a lower bound of 8 for association "Uses". However, these constraints are most effective when used to refine partial bound information provided by a designer. For instance, just by assuming a limit of 10 serial numbers, it is possible to infer that there is exactly one "Cutter" and one "Grinder", between 8 and 10 parts and at most 8 links among machines and parts.

Even if the constraints in the example seem trivial, a UML/OCL model may have many constraints like these and they can interact among them, making it impossible for users to consider all of them when choosing verification bounds. Hence, providing tool support to aid in the bound selection process can be helpful.



(a)

context Part **inv** UniqueSerials:
Part::AllInstances()−> isUnique(serial)

context AbstractMachine **inv** MachineAvailability:
Cutter::AllInstances()−> exists(c | c.ready) and
Grinder::AllInstances()−> exists(g | g.ready)

(b)

Figure 6.2: UML/OCL class diagram used as example. (a) diagram (b) OCL constraints

6.4 Constraint Propagation

As it was mentioned in Chapter 2, a solution to a CSP is an assignment of values to the variables such, that all constraints are satisfied. A typical approach for computing this assignment consists in searching it by assigning values to variables one at a time in a certain order and backtracking when a partial assignment cannot be extended any further. This search process is aided by "early evaluation" (detecting when a partial assignment is unfeasible and can be discarded) and "propagation" (removing values from the domain of unassigned variables using information about the constraints and the values of previously assigned variables). While solving a CSP is computationally expensive, propagation is much faster: practical implementations attempt to tighten domains in a pragmatic cost-effective way, instead of computing the optimal bounds with potentially slow computations.

As the goal of the technique presented in this chapter is tightening domain bounds rather than finding a specific instance within those bounds, propagation suits our needs better than CSP-solving. Thus, the CSP we define from the UML/OCL model under analysis will only be used to apply propagation. To this end, we will use the same CSP solver employed by EMFtoCSP, the ECL^{i}PSe Constraint Programming System[2] [120]. ECL^{i}PSe comes with a library called "IC" (hybrid integer/real Interval arithmetic Constraint solver) which provides powerful interval

2. http://eclipseclp.org

constraint propagation capabilities.

For example, when given a CSP with three variables X, Y and Z, each taking values in the integer interval $[-2, 10]$, and the constraints: $X = Y + Z$, $Y = \max(X, Z)$ and $X + 2Z \leq 2$, and applying constraint propagation techniques implemented by the IC solver, the original bounds are tightened as follows: $X \in [-2, 6], Y \in [-2, 6], Z \in [-2, 2]$. Two potential solutions for this CSP are $X = 0, Y = 0, Z = 0$ and $X = -2, Y = -1, Z = -1$, so it is clear that applying constraint propagation techniques may reduce search space boundaries while keeping the solutions of the CSP within.

Finally, it is important to remark that constraint propagation reduces the search space but in general it is unable to discard all unfeasible value assignments. For instance, in this example it cannot detect that there is no solution with $Z = 2$.

6.5 CSP Construction

6.5.1 Structure of the CSP

Table 6.1: Definition of the CSP used to tighten verification bounds

Variables (V)	**Domains** (D)	**Constraints** (C)
A variable cl for each class	Potential number of objects in class cl, either $[0, \infty)$ or a user-provided domain	– UML: generalizations, association end multiplicities, class multiplicities – OCL: all invariants – Correctness property under analysis, e.g. each class has at least one object
A variable as for each association	Potential number of links in association as $[0, \infty)$ or a user-provided domain)	– UML: association end multiplicities – OCL: invariants containing navigations through association as
A variable at for each attribute	Potential values of attribute at, depending on its data type, e.g. $[0, 1]$ for boolean, $(-\infty, \infty)$ for integers or a user-provided interval	– OCL: invariants accessing the value of attribute at
An auxiliary variable aux_e for each subexpression e in each OCL constraint	Potential values of the expression e depending on its data type. Non-basic types are abstracted, e.g. collections are abstracted as positive integers encoding their size.	– A constraint establishing the value of e in terms of the values of its subexpressions. – Correctness property under analysis, e.g. the root expression of each invariant must evaluate to 1 (all invariants must be true)

Considering the problem of establishing bounds for the verification of UML/OCL models, Table 6.1 defines the structure of a suitable CSP:

- The variables of this CSP will not characterize a complete instance, but rather the parameters that define the size of the verification space: how many objects and links and which attribute values should be considered when instantiating the model. Additional auxiliary variables are used for convenience to encode complex constraints associated with rich OCL expressions.
- The domains of this CSP will be the output of our approach, as we are addressing a bound tightening problem. The analysis may start without imposing any constraint at all on the domains, e.g. from 0 to ∞ objects in classes. In this way, this approach can attempt to automatically infer finite bounds for each of the variables in our problem. As this is usually not possible, the user may also define the set of bounds that he intended to use and let the constraint solver propagate the restrictions in order to tighten these bounds.
- The constraints of the CSP include graphical restrictions from the UML class diagram and the textual OCL invariants. In the case of OCL, the constraints are not a direct translation of the invariant (e.g., as done in [108]), but rather an abstraction of the invariant that only considers size information. The correctness property under analysis may also change the type of target instance that is being searched and therefore should also be included as a constraint.

6.5.2 Constraints for the UML Class Diagram

A UML model includes several implicit constraints that are depicted graphically, such as the multiplicity of associations. These constraints need to be included explicitly in the CSP. Our method follows the same approach employed in UMLtoCSP [45] and shared by EMFtoCSP, which is described in detail here [108]. Therefore, rather than repeating here the constraints for each class diagram element, we describe the resulting constraints for the example of Fig. 6.2:

$$\begin{aligned} \texttt{AbstractMachine} &= \texttt{Cutter} + \texttt{Grinder} && \text{Inheritance} \\ \texttt{Uses} &\leq \texttt{Part} * \texttt{AbstractMachine} && \text{Association} \\ \texttt{Uses} &= 4 * \texttt{AbstractMachine} && \text{Association end} \\ \texttt{Uses} &\leq \texttt{Part} && \text{Association end} \end{aligned}$$

6.5.3 Constraints for OCL Invariants

For the case of analyzing OCL expressions, we take advantage of the work of Yu et al. [119] on the verification of size properties of collection types, based on

abstracting away their contents but preserving the constraints on their sizes. This abstraction process transforms collection types into integer values representing the sizes of collections being replaced, and it is formally expressed in the form of an abstraction function α_T. In our case, we expand this abstraction process to cover, not only more operations involving collections, but also operations involving other data types, thus supporting the majority of operations described in the OCL specification [3]. Our abstraction function α_T is therefore slightly different to the one in [119], where t indicates the data type being abstracted: $t \in \{boolean\} \Rightarrow \alpha_T = t$

$t \in \{integer, unlimited\ natural, real\} \Rightarrow \alpha_T = t$
$t \in \{set, ordered\ set, bag, sequence\} \Rightarrow \alpha_T = integer$
$t \in \{string\} \Rightarrow \alpha_T = integer$ From this definition it can be seen, for example,

that strings, as it is the case for collections, are also transformed into integer values, that in this case represent their length.

Tables 6.2, 6.3, 6.4 and 6.5 summarize how an OCL invariant can be abstracted into a size constraint. Each table entry describes the translation of a specific type of OCL expression. The construction of the size constraint proceeds inductively over the structure of the OCL invariant: each subexpression of the invariant is matched with the appropriate table entry and produces a size constraint, whose value may depend on the size constraints of its subexpressions. The size constraint for the entire invariant is the one that corresponds to the root expression, which will be an expression of the form "Type::AllInstances()−>forAll(condition)".

In the following, we present the notation used to describe this process of abstraction, which is borrowed from [119]. Given an OCL expression e, we use $t(e)$ to denote the type of the value resulting from the evaluation of e. When abstracting such OCL expression, the size constraint is denoted as $e.c$, with $t(e.c) = boolean$, and it is expressed with the help of an auxiliary variable $e.v$ such that $t(e.v) = \alpha_T(t(e))$. That is, the auxiliary variable is of integer type when dealing with operations over collections or strings, and shares the expression data type otherwise. When $e.c$ holds, $e.v$ represents the size of the collection or the length of the string, for the case of operations on these data types; for the rest of data types, $e.v$ represents the result of evaluating e.

All the tables share the same structure: the second column represents the OCL expression being abstracted, the third column identifies the different combinations of data types that can occur in the context of that expression, and finally, the fourth column shows the size constraints derived for each particular case. The shorthand notation for data types in the second column denotes the real type as r, the integer type as i, the unlimited natural type as n, the boolean type as b, the string type as

3. http://www.omg.org/spec/OCL/2.3.1/

s, the set type as st, the ordered set type as os, the bag type as bg and the sequence type as sq. An asterisk ($*$) denotes the case when any data type can be used.

In the table entries, a special notation is used to refer to concepts in the CSP: $domain(v)$ is the set of potential values of variable v, $domain\_size(v)$ is the number of values in a domain, $num\_obj(T)$ is the CSP variable storing the number of objects of type T and $num\_links(A)$ is the CSP variable storing the number of links in association A.

Finally, when the information on the tables is applied to the analysis of the OCL constraints on the example of Fig. 6.2, the following constraints (simplified for readability) are obtained:

$$\begin{aligned} \texttt{Part} \leq domain\_size(\texttt{Serial}) \quad & \texttt{UniqueSerials} \\ \texttt{Cutter} \geq 1 \quad & \texttt{MachineAvailability} \\ \texttt{Grinder} \geq 1 \quad & \texttt{MachineAvailability} \end{aligned}$$

Table 6.2: OCL Operations on Numeric and Boolean Types

–	**OCL Expression** **e**	**Type** $t(e) : [t(e_1)][t(e_2)]$	**Size Constraint** **e.c**
1	*constant*	{r,i,n,b}: {r,i,n,b}	$e.v = constant$
2	$e_1 = e_2$	b: {r,i,n,b},{r,i,n,b}	$(e.v = (e_1.v = e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
3	$e_1 <> e_2$	b: {r,i,n},{r,i,n}	$(e.v = (e_1.v \neq e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
4	$e_1 < e_2$	b: {r,i,n},{r,i,n}	$(e.v = (e_1.v < e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
5	$e_1 \leq e_2$	b: {r,i,n},{r,i,n}	$(e.v = (e_1.v \leq e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
6	$e_1 > e_2$	b: {r,i,n},{r,i,n}	$(e.v = (e_1.v > e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
7	$e_1 \geq e_2$	b: {r,i,n},{r,i,n}	$(e.v = (e_1.v \geq e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
8	$e_1 + e_2$	{r,i,n}: {r,i,n},{r,i,n}	$(e.v = e_1.v + e_2.v) \wedge$ $e_1.c \wedge e_2.c$
9	$e_1 - e_2$	{r,i,n}: {r,i,n},{r,i,n}	$(e.v = e_1.v - e_2.v) \wedge$ $e_1.c \wedge e_2.c$
10	$e_1 * e_2$	{r,i,n}: {r,i,n},{r,i,n}	$(e.v = e_1.v * e_2.v) \wedge$ $e_1.c \wedge e_2.c$
11	e_1 / e_2	r: {r,i},{r,i}	$(e.v = e_1.v/e_2.v) \wedge e_2.v \neq 0 \wedge$

			$e_1.c \wedge e_2.c$
12	e_1 div e_2	i: {i,n},{i,n}	$(e.v = e_1.v \; div \; e_2.v) \wedge$ $e_2.v \neq 0 \wedge e_1.c \wedge e_2.c$
13	e_1 mod e_2	i: {i,n},{i,n}	$(e.v = e_1.v \; mod \; e_2.v) \wedge$ $e_2.v \neq 0 \wedge e_1.c \wedge e_2.c$
14	$-e_1$	r: r i: {i,n}	$(e.v = -e_1.v) \wedge e_1.c$
15	$e_1.abs()$	r: r i: i	$(e.v = \lvert e_1.v \rvert) \wedge e_1.c$
16	$e_1.floor()$	i: r	$(e.v = \lfloor e_1.v \rfloor) \wedge e_1.c$
17	$e_1.round()$	i: r	$(e.v = round(e_1.v)) \wedge e_1.c$
18	$e_1.max(e_2)$	{r,i,n}: {r,i,n},{r,i,n}	$(e.v = \max(e_1.v, e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
19	$e_1.min(e_2)$	{r,i,n}: {r,i,n},{r,i,n}	$(e.v = \min(e_1.v, e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
20	not e_1	b: b	$(e.v = 1 - e_1.v) \wedge e_1.c$
21	e_1 or e_2	b: b,b	$(e.v = \max(e_1.v, e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
22	e_1 and e_2	b: b,b	$(e.v = \min(e_1.v, e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
23	e_1 xor e_2	b: b,b	$(e.v = (e_1.v \neq e2.v)) \wedge$ $e_1.c \wedge e_2.c$
24	e_1 implies e_2	b: b,b	$(e.v = \max(1 - e_1.v, e_2.v)) \wedge$ $e_1.c \wedge e_2.c$
25	$e_1.attr$	{r,i,n,b,s}: {r,i,n,b,s}	$domain(e.v) \subseteq domain(attr)$
26	$e_1.toString()$	s: i	$((e_1.v = 0) \rightarrow (e.v = 1)) \wedge$ $((e_1.v > 0) \rightarrow$ $\quad (e.v = \lceil log_{10} \; e_1.v \rceil + 1)) \wedge$ $((e_1.v < 0) \rightarrow$ $\quad (e.v = \lceil log_{10} \; \lvert e_1.v \rvert \rceil + 2)) \wedge$ $e_1.c$
		s: n	$((e_1.v = 0) \rightarrow (e.v = 1)) \wedge$ $((e_1.v > 0) \rightarrow$ $\quad (e.v = \lceil log_{10} \; e_1.v \rceil + 1)) \wedge$ $e_1.c$
		s: r	$(e.v \geq 1)$
		s: b	$(e.v = 5 - e_1.v) \wedge e_1.c$

Table 6.3: Boolean Operations Over Collections

–	**OCL Expression** **e**	**Type** $t(e) : t(c_1)[t(c_2)\|t(e_1)]$	**Size Constraint** **e.c**
1	$c_1 = c_2$	b: {st,sq,bg},{st,sq,bg}	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v \neq c_2.v) \rightarrow$ $(e.v = 0)) \wedge$ $(((c_1.v = 0) \wedge$ $(c_2.v = 0)) \rightarrow$ $(e.v = 1)) \wedge$ $c_1.c \wedge c_2.c$
2	$c_1 <> c_2$	b: {st,sq,bg},{st,sq,bg}	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v \neq c_2.v) \rightarrow$ $(e.v = 1)) \wedge$ $(((c_1.v = 0) \wedge$ $(c_2.v = 0)) \rightarrow$ $(e.v = 0)) \wedge$ $c_1.c \wedge c_2.c$
3	$c_1 \rightarrow includes(o)$	b: {st,sq,bg}	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0) \rightarrow$ $(e.v = 0)) \wedge$ $c_1.c \wedge c_2.c$
4	$c_1 \rightarrow excludes(o)$	b: {st,sq,bg}	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0) \rightarrow$ $(e.v = 1)) \wedge$ $c_1.c \wedge c_2.c$
5	$c_1 \rightarrow$ $includesAll(c_2)$	b: {st,sq,bg},{st,sq,bg}	$(0 \leq e.v \leq 1) \wedge$ $((c_2.v > c_1.v) \rightarrow$ $(e.v = 0)) \wedge$ $((c_2.v = 0) \rightarrow$ $(e.v = 1)) \wedge$ $c_1.c \wedge c_2.c$
6	$c_1 \rightarrow$ $excludesAll(c_2)$	b: {st,sq,bg},{st,sq,bg}	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0 \wedge c_2.v \geq 1)$ $\rightarrow (e.v = 1)) \wedge$ $((c_2.v = 0) \rightarrow$ $(e.v = 0)) \wedge$ $c_1.c \wedge c_2.c$
7	$c_1 \rightarrow isEmpty()$	b: {st,sq,bg}	$(e.v = (c_1.v = 0)) \wedge$ $c_1.c$

8	$c_1 \rightarrow notEmpty()$	b: {st,sq,bg}	$(e.v = (c_1.v \neq 0)) \wedge$ $c_1.c$
9	$c_1 \rightarrow exists(e_1)$	b: {st,sq,bg},b	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0 \vee e_1.v = 0)$ $\rightarrow (e.v = 0)) \wedge$ $((e_1.v = 1) \rightarrow$ $(e.v = (c_1.v \geq 1))) \wedge$ $c_1.c \wedge e_1.c$
10	$c_1 \rightarrow forAll(e_1)$	b: {st,sq,bg},b	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0) \rightarrow$ $(e.v = 1)) \wedge$ $c_1.c$
11	$Type ::$ $allInstances()$ $\rightarrow forAll(e_1)$	b: st,b	$(0 \leq e.v \leq 1) \wedge$ $((num\_obj(Type) = 0) \rightarrow$ $(e.v = 1)) \wedge$ $((num\_obj(Type) > 0) \wedge$ $(e.v = 1) \rightarrow$ $(e_1.v = 1)) \wedge e_1.c$
12	$c_1 \rightarrow one(e_1)$	b: {st,sq,bg},b	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0 \vee e_1.v = 0)$ $\rightarrow (e.v = 0)) \wedge$ $((e_1.v = 1) \rightarrow$ $(e.v = (c_1.v \geq 1))) \wedge$ $c_1.c \wedge e_1.c$
13	$c_1 \rightarrow isUnique(e_1)$	b: {st,sq,bg}, {r,i,n}	$(0 \leq e.v \leq 1) \wedge$ $((e.v = 0) \rightarrow$ $(c_1.v \geq 2)) \wedge$ $(domain\_size(e_1.v)$ $\geq c_1.v) \wedge$ $c_1.c \wedge e_1.c$
		b: {st,sq,bg}, b	$(0 \leq e.v \leq 1) \wedge$ $((e.v = 1) \rightarrow$ $(c_1.v \leq 2)) \wedge$ $((e.v = 0) \rightarrow$ $(c_1.v \geq 2)) \wedge$ $(domain\_size(e_1.v)$ $(\quad \geq c_1.v) \wedge$ $c_1.c \wedge e_1.c$
		b: {st,sq,bg}, s	$(0 \leq e.v \leq 1) \wedge$

			$((e.v = 0) \rightarrow$ $(c_1.v \geq 2)) \wedge$ $c_1.c$

Table 6.4: Other Operations Over Collections

–	**OCL Expression** **e**	**Type** $t(e) : t(c_1)$ $[t(c_2)\|t(e_1)]$	**Size Constraint** **e.c**
1	$c_1 \rightarrow size()$	i: {st,sq,bg}	$(e.v = c_1.v) \wedge c_1.c$
2	$c_1 \rightarrow count(o)$	i: st	$(0 \leq e.v \leq 1) \wedge c_1.c$
		i: {sq,bg}	$(0 \leq e.v \leq c_1.v) \wedge c_1.c$
3	$c_1 \rightarrow max()$	{r,i}: {st,sq,bg}	true
4	$c_1 \rightarrow min()$	{r,i}: {st,sq,bg}	true
5	$c_1 \rightarrow product(c_2)$	i: {st,sq,bg}, {st,sq,bg}	$(e.v = c_1.v * c_2.v) \wedge$ $c_1.c \wedge c_2.c$
6	$c_1 \rightarrow flatten()$	i: {st,sq,bg}	$(e.v \geq c_1.v) \wedge c_1.c$
7	$c_1 \rightarrow sum()$	{r,i}: {st,sq,bg}	$((c_1.v = 0) \rightarrow$ $(e.v = 0)) \wedge c_1.c$
8	$c_1 \rightarrow asSet()$	st: st	$(e.v = c_1.v) \wedge c_1.c$
		st: {sq,bg}	$((c_1.v > 0) \rightarrow$ $(e.v > 0)) \wedge$ $(0 \leq e.v \leq c_1.v) \wedge$ $c_1.c$
9	$c_1 \rightarrow$ $asOrderedSet()$	os: st	$(e.v = c_1.v) \wedge c_1.c$
		os: {sq,bg}	$((c_1.v > 0) \rightarrow$ $(e.v > 0)) \wedge$ $(0 \leq e.v \leq c_1.v) \wedge$ $c_1.c$
10	$c_1 \rightarrow asBag()$	bg: {st,sq,bg}	$(e.v = c_1.v) \wedge c_1.c$
11	$c_1 \rightarrow asSequence()$	sq: {st,sq,bg}	$(e.v = c_1.v) \wedge c_1.c$
12	$\{constant\}$	{st,bg,sq}: {b,r,i,s}	$(e.v = 1)$
13	$c_1.navigation$	{st,bg,sq}: {st,bg,sq}	$(e.v \geq min\_mult(nav)) \wedge$ $(e.v \leq$ $max\_mult(nav) * c_1.v) \wedge$ $(e.v \leq$ $num\_links(Assoc(nav))) \wedge$

			$(e.v \leq$ $num\_obj(Type(nav))) \wedge$ $c_1.c$
14	$c_1 \rightarrow including(o)$	st: st	$(c_1.v \leq e.v \leq c_1.v + 1) \wedge$ $((c_1.v = 0) \rightarrow (e.v = 1)) \wedge$ $c_1.c$
		{sq,bg}: {sq,bg}	$(e.v = c_1.v + 1) \wedge c_1.c$
15	$c_1 \rightarrow excluding(o)$	{st,os,sq,bg}: {st,os,sq,bg}	$(\max(0, c_1.v - 1) \leq$ $e.v \leq c_1.v) \wedge c_1.c$
16	$c_1 \rightarrow union(c_2)$	st: st,st	$(\max(c_1.v, c_2.v) \leq e.v \leq$ $c_1.v + c_2.v) \wedge$ $c_1.c \wedge c_2.c$
		bg: {st,sq,bg},bg bg: bg,{st,sq,bg} sq: {sq,st},sq sq: sq,{sq,st}	$(e.v = c_1.v + c_2.v) \wedge$ $c_1.c \wedge c_2.c$
17	$c_1 \rightarrow$ $intersection(c_2)$	st: {st,bg},st st: st,{st,bg} bg: bg,bg	$(0 \leq e.v \leq \min(c_1.v, c_2.v)) \wedge$ $c_1.c \wedge c_2.c$
18	$c_1 - c_2$	st: st,st	$(\max(0, c_1.v - c_2.v) \leq$ $e.v \leq c_1.v) \wedge c_1.c \wedge c_2.c$
19	$c_1 \rightarrow$ $symmetricDiff(c_2)$	st: st,st	$(0 \leq e.v \leq c_1.v + c_2.v) \wedge$ $((c_1.v = 0) \rightarrow (e.v > c_2.v)) \wedge$ $((c_2.v = 0) \rightarrow (e.v > c_1.v)) \wedge$ $c_1.c \wedge c_2.c$
20	$c_1 \rightarrow append(o)$	os: os	$(c_1.v \leq e.v \leq c_1.v + 1) \wedge$ $((c_1.v = 0) \rightarrow (e.v = 1))$ $\wedge c_1.c$
		sq: sq	$(e.v = c_1.v + 1) \wedge c_1.c$
21	$c_1 \rightarrow prepend(o)$	os: os	$(c_1.v \leq e.v \leq c_1.v + 1) \wedge$ $((c_1.v = 0) \rightarrow (e.v = 1))$ $\wedge c_1.c$
		sq: sq	$(e.v = c_1.v + 1) \wedge c_1.c$
22	$c_1 \rightarrow insertAt(i, o)$	os: os	$(c_1.v \leq e.v \leq c_1.v + 1) \wedge$ $((c_1.v = 0) \rightarrow (e.v = 1))$ $\wedge c_1.c$
		sq: sq	$(e.v = c_1.v + 1) \wedge c_1.c$
23	$c_1 \rightarrow at(i)$	{b}: {st,sq,bg}	$(0 \leq e.v \leq 1)$
		{r,i}: {st,sq,bg}	true

		{st,sq,bg}: {st,sq,bg}	$(e.v \geq 0)$
24	$c_1 \rightarrow first(i)$	{b}: {st,sq,bg}	$(0 \leq e.v \leq 1)$
		{r,i}: {st,sq,bg}	true
		{st,sq,bg}: {st,sq,bg}	$(e.v \geq 0)$
25	$c_1 \rightarrow last(i)$	{b}: {st,sq,bg}	$(0 \leq e.v \leq 1)$
		{r,i}: {st,sq,bg}	true
		{st,sq,bg}: {st,sq,bg}	$(e.v \geq 0)$
26	$c_1 \rightarrow indexOf(o)$	i: {os,sq}	$(1 \leq e.v \leq c_1.v) \wedge c_1.c$
27	$c_1 \rightarrow reverse()$	os: os sq: sq	$(e.v = c_1.v) \wedge c_1.c$
28	$c_1 \rightarrow subOrderedSet(l, u)$	os: os	$(e.v = u.v - l.v + 1) \wedge c_1.c \wedge u.c \wedge l.c$
29	$c_1 \rightarrow subSequence(l, u)$	sq: sq	$(e.v = u.v - l.v + 1) \wedge c_1.c \wedge u.c \wedge l.c$
30	$c_1 \rightarrow select(e_1)$	st: st,b sq: sq,b bg: bg,b	$(0 \leq e.v \leq c_1.v) \wedge c_1.c$
31	$c_1 \rightarrow reject(e_1)$	st: st,b sq: sq,b bg: bg,b	$(0 \leq e.v \leq c_1.v) \wedge c_1.c$
32	$c_1 \rightarrow collect(e_1)$	st: st,* sq: sq,* bg: bg,*	$(e.v = c_1.v) \wedge c_1.c$
33	$c_1 \rightarrow closure(e_1)$	os: {os,sq},* st: {st,bg},*	$(e.v \geq c_1.v) \wedge (c_1.c)$
34	$c_1 \rightarrow any(e_1)$	*: {st,sq,bg},*	true
		b: {st,sq,bg},*	$0 \leq e.v \leq 1$
35	$c_1 \rightarrow sortedBy(e_1)$	os: {st,os},* sq: {bg,sq},*	$(e.v = c_1.v) \wedge c_1.c \wedge e_1.c$
36	$c_1 \rightarrow iterate(...)$	*:*	true
37	$Type :: allInstances()$		$e.v = num\_obj(Type)$

Table 6.5: OCL Operations on Strings

–	**OCL Expression** **e**	**Type** $t(e) : [t(s_1)][t(s_2)]$	**Size Constraint** **e.c**
1	$constant$	i: s	$e.v = strlen('constant')$
2	$s_1 = s_2$	b: s,s	$((e.v = 1) \rightarrow (s_1.v = s_2.v)) \wedge s_1.c \wedge s_2.c$
3	$s_1 <> s_2$	b: s,s	$((e.v = 0) \rightarrow (s_1.v = s_2.v)) \wedge s_1.c \wedge s_2.c$
4	$s_1 < s_2$	b: s,s	$0 \leq e.v \leq 1$
5	$s_1 > s_2$	b: s,s	$0 \leq e.v \leq 1$
6	$s_1 \leq s_2$	b: s,s	$0 \leq e.v \leq 1$
7	$s_1 \geq s_2$	b: s,s	$0 \leq e.v \leq 1$
8	$s_1 + s_2$	s: s,s	$e.v = s_1.v + s_2.v \wedge s_1.c \wedge s_2.c$
9	$s_1.concat(s_2)$	s: s,s	$e.v = s_1.v + s_2.v \wedge s_1.c \wedge s_2.c$
10	$s_1.substring(l, u)$	s: s	$e.v = u.v - l.v + 1 \wedge s_1.c \wedge u.c \wedge l.c$
11	$s_1.indexOf(s_2)$	i: s,s	$0 \leq e.v \leq s_1.v \wedge s_1.c$
12	$s_1.equalsIgnoreCase(s_2)$	b: s,s	$((e.v = 1) \rightarrow (s_1.v = s_2.v)) \wedge s_1.c \wedge s_2.c$
13	$s_1.at(i)$	s: s	$e.v = 1$
14	$s_1.size()$	i: s	$e.v = s_1.v \wedge s_1.c$
15	$s_1.characters()$	sq: s	$e.v = s_1.v \wedge s_1.c$
16	$s_1.toUpperCase()$	s: s	$e.v = s_1.v \wedge s_1.c$
17	$s_1.toLowerCase()$	s: s	$e.v = s_1.v \wedge s_1.c$
18	$s_1.toBoolean()$	b: s	$0 \leq e.v \leq 1$
19	$s_1.toInteger()$	i: s	$-10^{s_1.v} \leq e.v \leq 10^{s_1.v} \wedge s_1.c$
20	$s_1.toReal()$	r: s	$-10^{s_1.v} \leq e.v \leq 10^{s_1.v} \wedge s_1.c$

6.6 Experimental Results

In this section, we aim to evaluate the proposed method in order to answer the following questions:

- **Q1:** Is the execution time of the bound tightening procedure negligible with respect to the execution time of UML/OCL bounded verification?
- **Q2:** Does the bound tightening procedure reduce the execution time of UML/OCL bounded verification significantly?

Table 6.6: Input UML/OCL models.

Name	Classes	Assocs	Attrs	Inv
Teams [121]	5	3	6(0)	6
Company [122]	6	8	19(2)	16

6.6.1 Designing Experiments

To answer questions Q1 and Q2, we have considered two UML/OCL models where we attempt to validate whether the model is *strongly satisfiable*, i.e. it is possible to create an instance of each non-abstract class in the model. Table 6.6 summarizes some features of the models under analysis: the number of classes, associations, attributes (in parenthesis, boolean attributes) and invariants. Both models have been taken from lecture notes for software development courses (see references) as they tend to include a wide variety of UML/OCL features. Also, the models illustrate two levels of constraint density: "Teams" with few constraints and "Company" with many constraints, so a priori the second one should be harder to verify. Minor changes (e.g. rewriting association classes) were required to adapt the models to the particular syntax requirements of the verification tools.

For the sake of representativity, we are interested in measuring the performance of verification for both satisfiable and unsatisfiable models. As both examples are satisfiable, we have devised an unsatisfiable version of each one by adding one invariant that cannot be satisfied due to the rest of constraints.

We have measured the performance of the USE tool, in particular, the USE model validator plug-in [81]. This tool transforms the original UML/OCL model into relational logic formula to be checked using the KodKod relational solver [98], which relies on SAT-solvers like Sat4j[4]. The choice of this particular toolkit has been motivated by its popularity and its competitive execution time results.

The translation of the UML/OCL model into a CSP for propagation has been developed as an extension of the EMFtoCSP tool. Our proposed bound tightening procedure has been implemented using the interval solver IC from the ECL^{i}PS^e Constraint Programming System [120]. We have used this tool to compute the tightened bounds for our UML/OCL models, measuring the computation time and measuring the verification time in USE with the tightened bounds.

For each UML/OCL, different sets of input bounds have been considered in order to illustrate the performance of our approach in different scenarios. Three types of bounds are defined: the number of objects in each class, the number of links in each association and the potential values for integer attributes ([0,1] is trivially used for boolean attributes). For the sake of simplicity, we assume they are global for the entire model rather than having distinct bounds for each model element.

4. http://www.sat4j.org/

Table 6.7: Experimental results (I)

	Verification Bounds			USE-orig	
Name	**Class**	**Assoc**	**Attrib**	**Trans**	**Solv**
Teams (s)	$[1,5]$	$[1,10]$	$[0,300]$	0,5s	1,3s
	$[1,10]$	$[1,20]$	$[0,300]$	0,8s	2,9s
	$[1,15]$	$[1,30]$	$[0,300]$	1,3s	4,5s
Teams (u)	$[1,5]$	$[1,10]$	$[0,300]$	0,6s	0,2s
	$[1,10]$	$[1,20]$	$[0,300]$	0,8s	1,2s
	$[1,15]$	$[1,30]$	$[0,300]$	3,6s	4,0s
Company (s)	$[1,5]$	$[1,10]$	$[0,300]$	2,5s	233,4s
	$[1,10]$	$[1,20]$	$[0,300]$	5,1s	95,3s
	$[1,15]$	$[1,30]$	$[0,300]$	14,7s	1.464,8s
Company (u)	$[1,5]$	$[1,10]$	$[0,300]$	1,3s	903,4s
	$[1,10]$	$[1,20]$	$[0,300]$	3,4s	4.449,1s
	$[1,15]$	$[1,30]$	$[0,300]$	timeout (>10.000s)	

Settings Computer: HP EliteBook 8470p, Intel Core i7 3GHz 8Gb RAM
OS: Windows 7 Enterprise SP1 64 bits
Java: Java SE Runtime Environment 1.7
USE v3.06 , Solver Sat4j with bitwidth=32
ECL^{i}PS^e v6.1 64 bits

6.6.2 Results

Tables 6.7 and 6.8 summarize the results obtained in our experiments. Each entry in these tables contains the model being analyzed (s = satisfiable version, u = unsatisfiable version), and the initial verification bounds. Then, Table 6.7 shows the execution time (in seconds) for USE with the original bounds (**USE-orig**), and Table 6.8 shows the time spent by the bound tightening procedure (**Tight**) and the execution time for USE with the tightened bounds (**USE-tight**). Regarding the execution times for USE, we further identify the time required by the tool to translate the UML/OCL model into a formula (**Trans**) and the time needed by the solver to check the formula (**Solv**). Finally, we measure the ratio of improvement in the execution time (**Speedup**) in Table 6.8 as USE-orig divided by Tight+USE-tight (1 if there is no change, higher is better).

6.6.3 Discussion

As expected, the verification of the "Teams" model is faster than the verification of "Company". Model size (less associations and attributes) and the number of invariants (6 vs 16) are the reasons for this difference.

Regarding question **Q1** (is the time spent tightening bounds negligible?), Table 6.8 shows that bound tightening requires less than one second in every example. Thus, the overhead generated by bound tightening will only be noticed in those

Table 6.8: Experimental results (and II)

	Verification Bounds				USE-tight		
Name	**Class**	**Assoc**	**Attrib**	**Tight**	**Trans**	**Solv**	**Speedup**
Teams (s)	$[1,5]$	$[1,10]$	$[0,300]$	0,8s	0,4s	1,3s	x0,76
	$[1,10]$	$[1,20]$	$[0,300]$	0,8s	0,8s	5,7s	x0,50
	$[1,15]$	$[1,30]$	$[0,300]$	0,8s	1,2s	5,3s	x0,79
Teams (u)	$[1,5]$	$[1,10]$	$[0,300]$	0,8s	0,3s	0,3s	x0,50
	$[1,10]$	$[1,20]$	$[0,300]$	0,8s	0,8s	1,3s	x0,69
	$[1,15]$	$[1,30]$	$[0,300]$	0,8s	1,3s	2,9s	x1,53
Company (s)	$[1,5]$	$[1,10]$	$[0,300]$	0,8s	0,9s	18,8s	x11,54
	$[1,10]$	$[1,20]$	$[0,300]$	0,8s	3,6s	56,9s	x1,64
	$[1,15]$	$[1,30]$	$[0,300]$	0,8s	8,6s	249,1s	x5,94
Company (u)	$[1,5]$	$[1,10]$	$[0,300]$	0,8s	0,9s	16,2s	x50,42
	$[1,10]$	$[1,20]$	$[0,300]$	1,0s	4,9s	2.081,3s	x2,13
	$[1,15]$	$[1,30]$	$[0,300]$	1,0s	10,5s	4.414,6s	–

Settings Computer: HP EliteBook 8470p, Intel Core i7 3GHz 8Gb RAM
OS: Windows 7 Enterprise SP1 64 bits
Java: Java SE Runtime Environment 1.7
USE v3.06 , Solver Sat4j with bitwidth=32
ECL^{i}PS^e v6.1 64 bits

examples where the solver verifies the model in a couple of seconds.

With respect to **Q2** (does bound tightening reduce verification time?), the answer is similar to the previous one: the effect of bound tightening is most noticeable in models where verification is most complex. For those examples, significant reductions can be achieved with some examples running 50 times faster or using 40 minutes less CPU time.

In one specific unsatisfiable example, tightening allows the verification to complete without a timeout. This is a very positive result, as being able to check the model with larger bounds allows designers to gain more confidence in the answer from the bounded verification tool.

Again, for "easy" models that can be verified quickly, bound tightening may fail to cause any reduction at all or it may be insufficient to compensate the bound tightening overhead. In any case, the performance gains in "hard" instances compensate this small penalty in "easy" instances.

To sum up, the performance gains offered by the bound tightening procedure on UML/OCL model verification will depend on a variety of factors:

– *The original domain bounds:* If the initial domains are very small, the verification is typically fast and the speedup provided by this technique might not be noticeable. For larger domains, the reduction in execution can become significant.
– *The existence of a witness:* If the correctness property has a witness (an ex-

ample or counterexample) within the bounded domain, verification can stop as soon as it is found, without having to explore the entire domain. In this scenario, reducing the size of the bounded domain may have little impact in the verification time. In contrast, bound tightening will be most effective when there are no witnesses or the verification space contains a low rate of witnesses.

– *The number (and strictness) of constraints in the model:* If the model has few or weak constraints (e.g. multiplicities "0..*"), this approach may fail to reduce bounds in a noticeable way. Conversely, tightening will be most effective for highly constrained models.

That is, the benefits of our approach are most noticeable in the models that take longer to verify, which are the ones where the user can benefit most from a speedup. Furthermore, in models where bound tightening does not produce any speedup, the overhead it incurs is negligible. For these reasons, bound tightening can be a valuable addition to any bounded verification framework for UML/OCL.

6.7 Conclusions

In this chapter, we have introduced a novel technique to improve the bounded verification of UML/OCL models. This approach aims to assist users in the selection of verification bounds, a task which currently lacks adequate tool support.

The proposed method operates by translating the UML/OCL model into a CSP that captures the graphical UML restrictions and the textual OCL invariants. The translation abstracts all information which cannot be used to infer domain bounds. Then, interval constraint propagation techniques are used to tighten the domain bounds. Applying constraint propagation is much faster than verifying the model and the smaller bounds can reduce the verification time significantly. This speedup has been shown experimentally in several models verified using the SAT-based USE tool.

This approach can be used in two different ways: as a preprocessing stage before verification, or as part of an interactive process to guide the choice of bounds.

Using Static Verification Tools For Testing Model Transformations

7

Landscape of Model Transformation Testing Approaches

After discussing several mechanisms to improve static model verification tools, in Part III the focus is on how these tools, and in particular EMFtoCSP, can be used at the time of testing model transformations, one of the key elements of MDE-based software development approaches.

As it was the case in Part II, the first chapter of Part III is devoted to present the state of the art on the matter. To begin with, some generalities about model transformation testing are exposed, and right after that, existing approaches, as of this writing, are analyzed. Finally, some areas of improvement are identified and discussed. How to address some of them will be the objective of the rest of chapters in this part.

7.1 Generalities About Model Transformation Testing

Writing model transformations is a delicate, cumbersome and error-prone task. In general, MDE-based processes are very sensitive to the introduction of defects. A defect in a model or a model transformation can be easily propagated to the subsequent stages, thus causing the production of faulty software. This is especially true when developing systems of great size and complexity, which usually requires writing large chains of complex model transformations.

In order to alleviate the impact defects can cause, a great deal of effort has been

made to find mechanisms and techniques to increase the robustness of MDE-based processes. Thus far, these efforts have been centered on trying to somewhat adapt well-known approaches, such as testing or verification, to the reality of models and model transformations of MDE (see [123], [124] or [125] for recent surveys). This has resulted in the appearance of a series of testing and verification techniques, specifically designed to target model transformations.

The techniques available to test a model transformation can be classified in the same way that those employed in, let's say, more traditional testing approaches. That is, static analysis techniques like code inspections or walkthroughs can be used to review the model transformation source code without having to executed it, but dynamic analysis techniques requiring model transformation execution can be employed as well.

In the rest of the chapter, when talking about test model generation, we will focus on this second group. When dynamic analysis techniques are applied, the methodology is essentially the same followed in traditional testing approaches. That is:

- Determine the adequate input test data to test the model transformation with.
- Run the model transformation with the input data obtained from the first stage.
- Analyze the outputs yielded by the model transformation execution to detect the presence of errors.

Again, different strategies can be followed at the time of conducting the testing process, being black-box and white-box approaches the most popular ones. Black-box approaches generate input test data (henceforth test models) out of the analysis of the model transformation specification, and white-box approaches do the same out of the analysis of the model transformation internals. As it is the case when testing software developed using "more traditional" means, mixed strategies like the one sketched in Fig. 7.1, are encouraged. However, and independently of the strategy followed, testers need a mechanism to decide which elements (either from the specification in the case of black-box approaches, or from the model transformation internals in the case of white-box approaches) will be the focus of testing. This mechanism, which is typically known as "test adequacy criteria" or "coverage criteria", serves two purposes. It steers the test model generation process, and determines the desired intensity of the testing efforts. The expression "coverage criteria" comes from the fact that steering the test model generation process implies determining which parts of the specification (in the case of black-box approaches), or which parts of the model transformation internals (in the case of white-box approaches), will be "covered" by the generated test models. Obviously, the desired scenario here is to achieve a 100% degree of coverage, but normally this is not possible because of practical reasons. A test adequacy criterion can also be seen as a

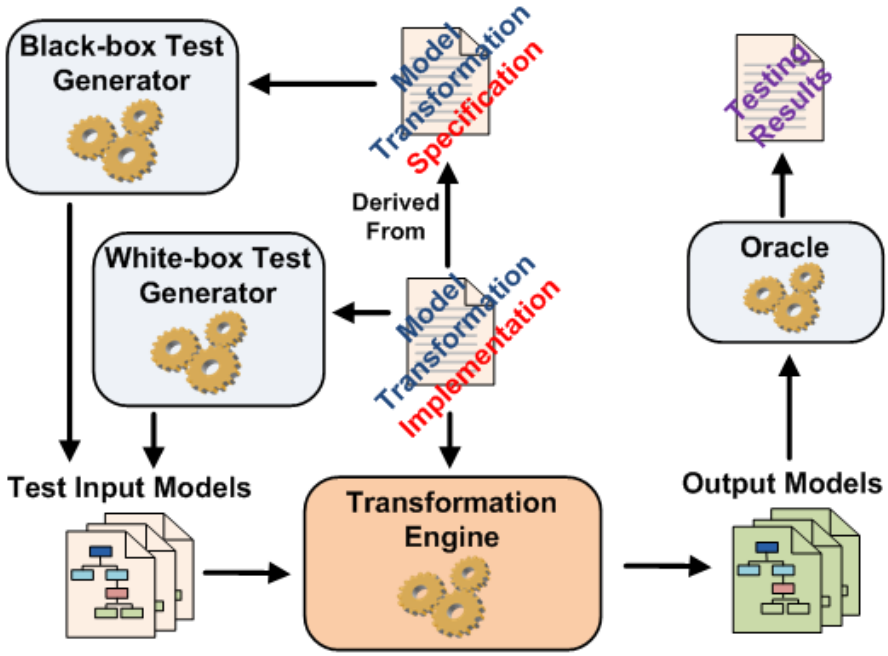


Figure 7.1: Mixed approach to model transformation testing

stopping rule for the testing process [126].

When it comes to the third step, the analysis of the output to uncover errors, it is typical the utilization of oracles. As stated in Chapter 2, an oracle is any program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object [35]. In the case of model transformation testing, oracles are usually implemented in the form of model comparison approaches, in those cases where the expected output model is available, or, if that is not the case, by means of some kind of "contract" that validates the output with respect to an existing specification, or a set of post-conditions the output model must fulfill.

In the following section, the focus is on the description of existing approaches addressing the problem of test model generation. The same is done immediately afterward, with those approaches focusing on the construction of oracles.

7.2 Generation of Test Models

7.2.1 Black-box Approaches

The majority of existing test model generation approaches follow a black-box strategy where the model transformation specification is somehow analyzed to steer the generation process. In particular, in these cases is typical to analyze the model transformation's input metamodel, with the intent of generating a set of test models representative of its instance space, something known as metamodel coverage. The problem though, is that a metamodel's instance space is usually infinite, so what the majority of these methods really do is to use partition analysis [127] to identify

non-empty and disjoint regions of the instance space where models share the same features, and then, create one test model out of each region identified.

Although not related to model transformation testing, to the best of our knowledge, the first attempt of using partition analysis to derive test models out of UML class diagrams was made by Andrews et al. [128]. In this work, partition analysis is employed to identify representative values of attributes and association ends multiplicities to steer the generation of test models. The work of Andrews et al. served as inspiration for the black-box test model generation approach proposed by Fleurey et al. [129] where the partition analysis of [128] is used to identify representative values of the model transformation's input metamodel. Once these representative values are found, they are used during the test model generation stage. Test models are generated by means of an adaptation of a genetic algorithm called "bacteriologic algorithm" [130, 131]. The authors also propose an enhancement of the process based on the concept of "effective metamodel". They coined this expression to make reference to those sections of the model transformation's source and target metamodels that are really relevant for the transformation. The idea is that, if the effective metamodel is provided as an input for the test model generation process, then non-relevant sections of the model transformation's input metamodel will not be considered during the process.

The work of Fleurey et al. influenced a number of proposals in this field. In [132], Brottier et al. proposed a variation of the approach described in [129] based on the utilization of "model fragments". Model fragments are, in the authors' words, "interesting object structures" from the model transformation's input metamodel, that are worth being covered by the generated test models. The proposed approach assumes that these model fragments are provided by the tester. Later on, Fleurey et al. [133] combined the concepts in [129] and [132] to develop a framework called Metamodel Coverage Checker (MMCC) [1] aimed at assessing the quality of the generated test models, based on the analysis of different coverage criteria.

Also influenced by [129] and [128] is the work of Maher Lamari [134] which proposes a formal language called MTSpecL for the specification of model transformations. This language is accompanied by a tool that allows computing effective metamodels for the model transformation's source and target metamodels, as well as representative values. This information is used during the test model generation stage, which is also supported.

Another black-box mechanism was proposed by Sen et al. [135]. An interesting feature of this approach is that the generation of test models is based on the resolution of a SAT problem by means of Alloy [97] [2]. In this approach, different

1. http://www.irisa.fr/triskell/Software/protos/MMCC
2. http://alloy.mit.edu/alloy/

types of constraints expressed in first-order relational logic are derived out of the analysis of model transformation specifications. These constraints are then used to build a boolean satisfiability problem, which is solved by means of a SAT solver at the time of generating the test models. More particularly, these constraints are derived from the model transformation's input metamodel, a set of model transformation pre-conditions written in OCL, and model fragments obtained by using the tool MMCC mentioned before. It is also possible to define additional constraints expressed directly as first-order relational logic predicates, to better adjust the test model generation process. The approach is supported by the presence of a tool called "Cartier". This work was complemented with an approach [136], based on mutation analysis [137], where generated test models are evaluated in terms of their efficiency at the time of uncovering bugs.

Also based on the utilization of a logic encoding is the approach proposed in [138]. In this work, the authors focus on the presentation of a constructive logic encoding of metamodels described in the MOF[3] language. At the time of generating test models, the encoded metamodel, along with a model transformation pre-condition expressed in the same constructive logic, are fed into a solver. Some experiments using Prolog are shown as a proof of the feasibility of the approach.

Continuing with black-box test model generation approaches based on the utilization of some kind of solver, [139, 140] propose a mechanism based on the utilization of UMLtoCSP [45] for the generation of test models. In this approach, a visual language called PAMOMO (Pattern-based Model-to-Model Specification Language) [141], member of a family of modeling languages presented in [142], is employed to write the specification of a model transformation. That specification may contain pre-conditions that the model transformation's input model must hold, post-conditions that the model transformation's output model must hold, and invariants, establishing some kind of relationship between model transformation's input and output models. Once the specification has been written, the next step is to generate a number of OCL assertions out of it. These OCL assertions along with the model transformation's input metamodel constitute the constraints that every generated test model must fulfill. These constraints are then complemented with an additional OCL expression built according to a specific coverage criteria to be chosen among the seven proposed. This OCL expression, along with the ones derived from the specification, and the input metamodel, are then fed to UMLtoCSP for the generation of one test model. UMLtoCSP will check if it is possible to build a model conforming the model transformation's input metamodel and holding at the same time the additional OCL constraints. If so, the result will be a test model fulfilling all these constraints. More test models can be obtained if the process is repeated a number of times, each time making some variation during the application

3. http://www.omg.org/spec/MOF/

of the coverage criteria of choice, so that the resulting OCL expression differs from the ones obtained in previous executions of the method.

To finish with black-box approaches, Vallecillo et al. [143, 144] presented a proposal based on the concept of "Tract" (a generalization of the concept of model transformation contract [145, 146]), where test models are generated by means of a language called ASSL, part of the USE tool [4], taking advantage of the constraints defined in these tracts. This proposal has been expanded by Wimmer and Burgueño [147, 148], to also cover the testing of model-to-text and text-to-model transformations.

7.2.2 White-box Approaches

Compared to the number of black-box test model generation proposals, the number of existing white-box approaches is rather small.

Fleurey et al. [129] propose a white-box enhancement of the black-box approach presented at the beginning of the previous subsection. Essentially, the only variation is that the method now performs a static analysis of the model transformation to compute its effective metamodel and representative values. This information is then used to generate the test models. The method is described at a high-level of abstraction, without focusing on any model transformation language.

Küster et al. [149] also propose three different white-box based testing techniques. However, these techniques rely on the analysis of model transformations expressed as a high level and non executable semi-formal description built using the IBM WebSphere Business Modeler [5]. In the first technique, model transformations rules are analyzed to create metamodel templates, that are automatically instantiated to generate suitable test models. The second technique generates test models with the intent of exercising well-formedness constraints, defined in the model transformation's input metamodel, that can be violated by the interplay of several transformation rules. More specifically, the process consists in finding those model elements from the model transformation's input metamodel that are sensitive to the model transformation rules, and then identifying the well-formedness constraints that apply on these model elements. For each of these constraints, a new test model is built. Finally, the third approach proposed consists in the analysis of pairs of rules to construct test models that lead to the detection of confluence errors.

Wang et al. [150] also propose a tool for the automatic generation of test models following a white-box approach. The work builds on the concepts introduced by [129] (effective metamodel and representative values) and presents a tool which is capable of identifying effective metamodels and representative values out of the

4. http://sourceforge.net/projects/useocl/
5. http://www.ibm.com/developerworks/downloads/ws/wbimod/

analysis of model transformation rules expressed in the Tefkat transformation language [6]. This information is used during the test model generation stage, which is also supported.

White-box techniques can also be used in coverage analysis, to measure the quality of the generated test models. Regarding this, [151] proposes a number of white-box coverage measures for ATL transformations, namely rule coverage, instruction coverage and decision coverage, that are used to check how a number of test models cover ATL transformations. Although no generation of test model is proposed, the approach could be useful to check the quality of the tests generated with any other approach, especially for model transformations where the designer may want to limit the number of test models generated.

7.3 Oracle Construction

After the description of approaches devoted to the generation of test models, in this section the focus is on those ones devoted to the construction of oracles.

The importance of an oracle lies on the fact that it is the element that allow testers to find out whether the testing experience uncovered any errors or not. As it was mentioned before, oracles usually take the form of model comparison approaches, or mechanisms to check whether the output model satisfies a series of desired constraints. In this second case, these constraints are normally derived from some kind of model transformation "contract" where pre- and post-conditions about the models involved in the model transformation are stated. These oracles are typically referred to as "partial oracles".

Some of the approaches devoted to the generation of test models, described in the previous section, also propose mechanisms for the construction of oracles. In the majority of cases, they build model transformation contracts that are then exploited to derive constraints that model transformation's output models must fulfill. For example, Maher Lamari [134] claims that the formal language MTSpecL for the specification of model transformations, is also useful for the generation of invariants relevant for target models, or for both, source and target models combined. Unfortunately, how these invariants would be generated is not described in detail. Similarly, the work of Fiorentini et al. [138] also covers succinctly how their approach can be expanded to check whether output models satisfy model transformation post-conditions. Analogously the proposal of Vallecillo et al. [143, 144] based on the concept of "Tract", can be used to check the presence of errors in model transformations. Tracts define, among others, constraints on output models, and on the relationship among input and output models. These constraints, along with

6. http://tefkat.sourceforge.net/

the ones in the model transformation's output metamodel, can be used to check the correctness of output models by means of the USE tool. Finally, the approach of Guerra et al. [139, 140] also supports the construction of oracles. In this case, the constraints that model transformations must fulfill are expressed using the visual language PAMOMO, which can be compiled into OCL [141].

Among the few specific approaches devoted to the construction of oracles, Cariou et al. [152] propose a mechanism to check the correctness of a model transformation with respect to model transformation contracts expressed in OCL. However, the authors put the focus on detailing how these contracts are built [153], rather than on explaining how to build the oracle itself, out of the constraints in the contract. In their words, once the contract is built, it suffices to use a standard OCL evaluator on the model transformation's output model to check whether the constraints in the contract are fulfilled or not.

Mottu et al. have also explored the oracle construction issue. In [154] they briefly describe at a high level of abstraction up to 6 different techniques to build an oracle. The same authors have also presented a proposal for the enhancement of model transformation contracts expressed in OCL, by means of mutation analysis. [155].

As it was mentioned before, oracles relying on the notion of model transformation contract are usually known as partial oracles, since they are only valid to check whether the output model fulfill a certain number of properties, normally expressed in the form of constraints. However, this is not the only strategy available to build oracles. For example, model comparison techniques has been identified as a major task in model transformation testing by [156].

Lin et al. [157] propose a testing framework integrated with the C-SAW model transformation engine, equipped with model comparison capabilities. In this framework, the transformation language is an extension of OCL called Embedded Constraint Language (ECL). When an ECL model transformation is tested, and given that the expected output model must be provided, the tool is capable of highlighting the differences between the actual and expected output models.

Finot et al [158] also propose using model comparison techniques when only a part of the expected model is available. In this approach, when a model transformation is tested, the actual output model is compared to a partial representation of the expected output model. In order to make it work, the tester must provide, apart from the partial representation of the expected output model, a set of patterns indicating which parts of the output model must be ignored during the comparison. The rationale behind this is simple. The partial representation of the expected output model is considered the part of the model that the tester is capable to predict. The patterns represent the rest of the model, that is, the unpredictable part that should not be considered when running the oracle. The authors also provide an implementation

of the approach, where EMFCompare [7] is used to carry out the model comparison.

EMFCompare is also used in the EUnit testing framework [159] for model transformations written in the Epsilon Transformation Language (ETL) [8]. In this framework, the philosophy to test a model transformation is similar to that of the JUnit testing framework [9]. A similar approach is proposed in [160], although in this case, the expected output model is transformed into assertions and the actual one is checked against them.

Finally, in [161], instead of implementing a JUnit-like framework, the proposal is to extend and adapt JUnit itself to support the testing of model transformations serialized in XML according to the XMI specification [10].

7.4 Challenges and Areas of Improvement

After having introduced the state of the art, in this section we will enumerate some of the challenges identified.

The first challenge has to do with the nature of the data involved in the testing process. Models tend to be complex and large structures conforming to metamodels that, in their turn, are also large and complex, and are possibly enriched with well-formedness rules expressed in some kind of constraint language, like OCL. This complexity affects test data generation and oracle construction mechanisms.

In the particular case of test model generation, model complexity turns it into a constraint solving problem, because it normally implies searching for a graph-like structure satisfying a, probably large, number of constraints in a, also probably large, search space. Admittedly, the number of existing approaches devoted to tackle this problem is not precisely large.

Additionally, an important number of test model generation approaches based on the utilization of black-box strategies, are also based on the utilization of the partition analysis technique proposed by Andrews et al. [128]. The problem with this is that this technique only considers OCL constraints superficially at the time of building partitions. This means that, approaches influenced by this work do not typically exploit properly the presence of OCL constraints in the model transformation's input metamodel, thus ignoring a valuable source of information that could help to achieve a better coverage. Unfortunately, the utilization of constraint or SAT solvers during test model generation is not free of inconveniences, either. Actually, this turns the test model generation problem into the same kind of problem studied in Part II, devoted to static model verification approaches, so challenges discussed

7. http://www.eclipse.org/emf/compare
8. http://www.eclipse.org/epsilon/
9. http://junit.org/
10. http://www.omg.org/spec/XMI/

there apply here as well.

There is also room for improvement when it comes to test model generation approaches based on the utilization of white-box strategies. The first obvious problem is that the number of existing approaches is really small. Moreover, the coverage criteria employed in these approaches is the same that is typically used in black-box approaches, that is, the coverage of the model transformation's input metamodel. There is nothing wrong with this, but since these kind of approaches are based on the analysis of model transformation internals, they should take advantage of this to try to cover the model transformation itself. After all, since there are already black-box approaches for covering the model transformation's input metamodel, it is always possible to combine different testing strategies to maximize the degree of coverage of the different artifacts involved in a model transformation. Finally, it is also important to mention that the existing approaches do not focus on popular model transformation languages like ATL or QVT.

When it comes to oracle construction, apart from the challenges derived from the complex nature of models, there are some others that are worth mentioning. Starting with approaches based on model comparison, it is obvious that is not always possible to have the expected output model available. And, even though if that model is available, model comparison is a complex problem in itself, equivalent to computing the graph isomorphism problem, which is a well-known NP-Complete problem [162]. This makes difficult to conduct model comparison in an efficient manner.

Oracle construction approaches based on the utilization of some kind of contract also present some inconveniences. The first one has to do with the construction of the contracts themselves, since stating the relationship between input and output models can be as complex and error-prone as writing the model transformation itself. Another problem is that these approaches do not normally check the output model exhaustively, but just the compliance to a number of desired properties. Moreover, and despite the number of properties checked, the problem in this case is, again, similar to the one discussed in Part II.

To finish, the last challenge we would like to mention has nothing to do with the approaches mentioned here, but with the current level of maturity of MDE tools. Compared to state-of-the-art tools employed in traditional software development, MDE tools do not show the same level of sophistication. Moreover, the integration of the available tools can be difficult or directly impossible to achieve. In general, MDE tools improve slowly, but still, more and better tools are needed. The presence of better tools may also facilitate the appearance of new and more effective testing approaches.

Our enhancement proposals are based on the test model generation stage. In particular, we propose two test model generation approaches, one following a black-

box strategy, and the other following a white-box one. Our black-box approach is based on conducting a partition analysis out of the analysis of the OCL constraints in the model transformation's input metamodel. As it was mentioned before, existing approaches based on partition analysis only exploit OCL constraints superficially. The white-box proposal tries to maximize the coverage of ATL model transformation internals. Existing white-box approaches neither focus on maximizing the coverage of model transformation internals nor cover popular model transformation languages like ATL. In both cases, the EMFtoCSP tool presented in Chapter 4 is used for the actual generation of test models.

7.5 Conclusions

In this chapter the current state of the art on model transformation testing has been presented. The different approaches have been described, some areas of improvement have been identified, and some enhancement proposals have been made. In the rest of chapters of Part III we will delve more into these proposals, starting with our black-box approach for the generation of test models.

8

A Black-box Test Model Generation Approach Based on Constraint and Partition Analysis

8.1 Motivation

Category-partition testing [127] consists in partitioning the input domain of the element under test, and then selecting test data from each class in the partition. This is a technique that an important number of the existing black-box test model generation approaches apply over the model transformation's input metamodel. The challenge when using partition analysis, though, is building the best partition possible. Since one test model is usually created out of each region identified, partitions should be small enough, so that all the models from the same region are as homogeneous as possible (meaning that, the sample model from that region can be used to represent all models from that same region, thus reducing the number of test models needed to get a sufficient confidence level on the quality of the transformation). Existing approaches address this by taking advantage of the fact that input metamodels usually come in the form of UML class diagrams complemented with constraints expressed in OCL. Therefore, partition analysis focuses on elements like association multiplicities, attributes values or OCL constraints to partition the model. However, in this last case, current approaches tend to be very superficial, either focusing only on simple OCL constraints, or deriving just obvious regions that do not require a deep analysis. This limits the representativeness of the generated test models and also the degree of coverage achieved when dealing with non-trivial metamodels.

As an example of this, Fig. 8.1(a) shows a metamodel describing the relationship between research teams and the papers they submit for publication. A simple partition analysis would try to exploit the presence of a numerical value in the OCL invariant stating that every team must have more than 10 submissions accepted. However, that alone is not enough to generate an interesting partitioning. A more fine-grained analysis of the constraint would reveal that beyond testing the transformation with teams with more than 10 accepted submissions, the transformation should also be tested using an input model with teams with more than 10 accepted submissions, and at least one rejected one. This conclusion is reached by analyzing the "select" condition in the OCL expression (more details on this later on). Fig. 8.2 shows the difference in the output produced by both analyses. Obviously, the second one exercises more the transformation and therefore may uncover errors not detected when using only the first one.

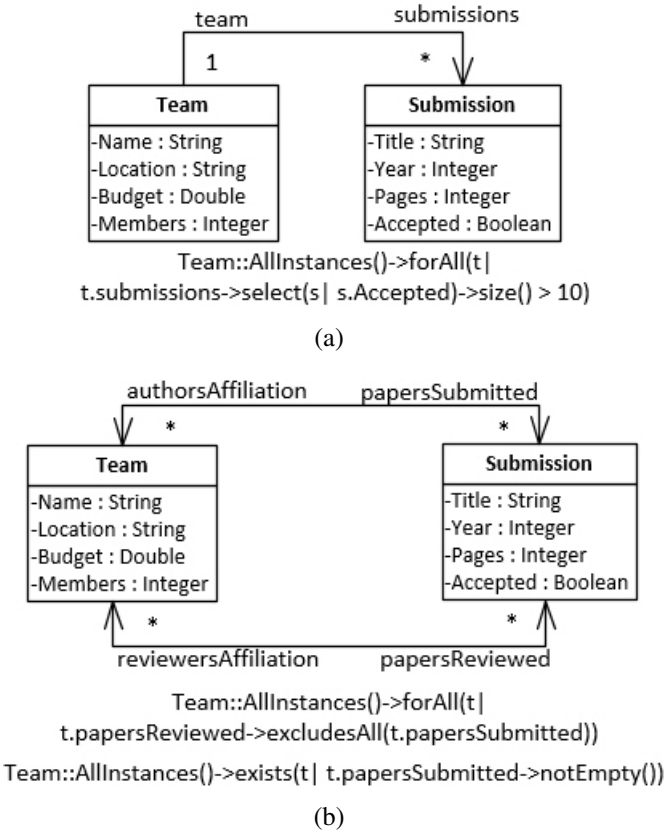


Figure 8.1: Two versions of the metamodel for the examples used throughout the paper.

In this chapter, we present a mechanism for the generation of input test models based on a combination of constraint and partition analysis over the OCL invariants of the model transformation's input metamodel. The method covers a substantial amount of OCL constructs and offers up to three different test model generation modes. Besides, it can be used in isolation, or combined with other black-box or

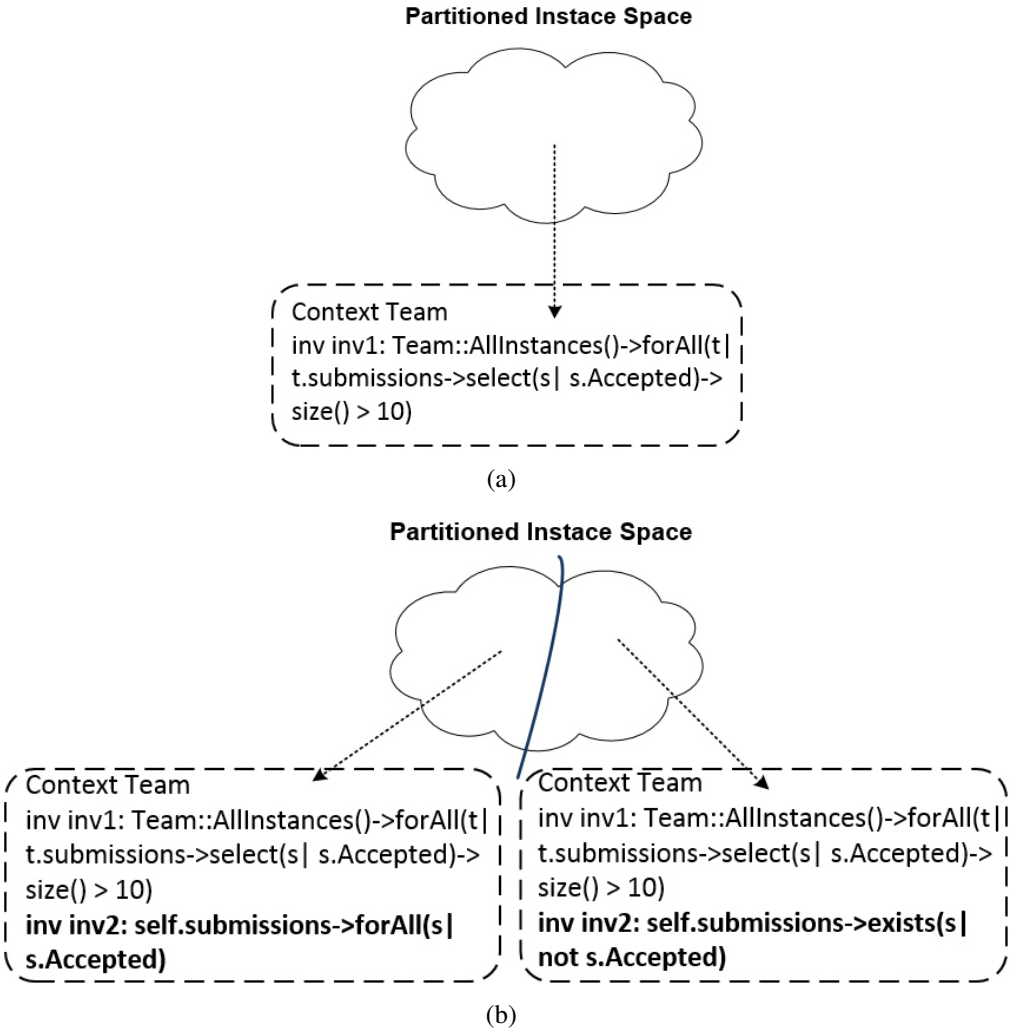


Figure 8.2: Results of two different partition analyses over the metamodel example.

white-box approaches to enhance the testing experience.

8.2 The Method in a Nutshell

Fig. 8.3 depicts how our method works. The model transformation's input metamodel characterizes a certain domain, and its instance space, possible inputs for the transformation. In the figure, dashed arrows indicate what characterizes certain elements, whereas solid arrows are data flows. When generating test models, the component called "OCL Analyzer" partitions the metamodel's instance space by analyzing its OCL invariants. As a result, a series of new OCL invariants characterizing the regions of the partition are obtained. This information, along with the input metamodel is then given to the "Test Model Generator" component, based on the EMFtoCSP tool, for the actual creation of the test models.

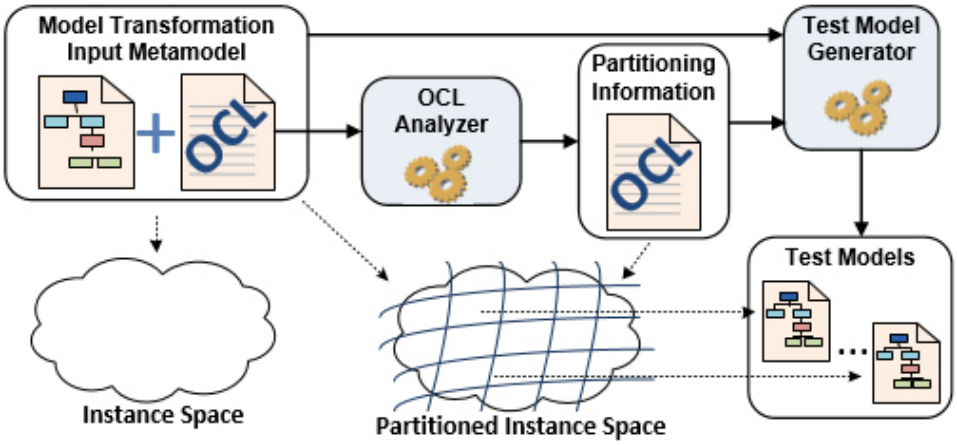


Figure 8.3: Overall picture

8.3 OCL Analysis

In this section, we begin the description of how to identify partitions in the input metamodel's instance space, focusing on the first step: analyzing the OCL invariants in the input metamodel to generate new OCL invariants characterizing suitable regions of the instance space. Next section uses these constraints to create the actual partitions.

Firstly, we talk about the OCL constructs supported by the method. After that, we describe how to systematically analyze complex OCL invariants made up by arbitrary combinations of the supported constructs.

8.3.1 OCL Constructs Supported

The supported OCL constructs have been classified in five groups and presented here in tabular form. The first group corresponds to expressions involving the presence of boolean operators (Table 8.1). The second group is about expressions formed by a boolean function operating over the elements of a collection (Table 8.2). The third group includes those boolean expressions involving the presence of arithmetic operators (Table 8.3). The fourth group contains other non-boolean expressions, that can be part of more complex boolean expressions (Table 8.4). Finally, the last group (Table 8.5) shows equivalent expressions for boolean expressions from Tables 8.1, 8.2 and 8.3 when they are negated.

Tables 8.1, 8.2, 8.3 and 8.4 share the same structure. For any given row, the second column contains a pattern. Analyzing an OCL invariant implies looking for these patterns, and every time one of them matches, the information in the third column indicates how to derive new OCL expressions characterizing suitable regions in the instance space. A dash (-) indicates that no new OCL expressions are derived. The rationale behind a given pattern and the expressions in the "Regions" column is simple: the pattern represents the invariant that the model must hold, and the

information in the "Regions" column are more refined expressions that must also hold when the pattern holds. For example, the entry 1 in Table 8.1 indicates that the pattern expression holds if the two subexpressions evaluate to the same value. The subexpressions in the "Regions" column indicate that there are two possibilities for this: either both are true, or both are false.

Table 8.5 is slightly different, though, and that has to do with how the method deals with negated expressions. Each time a negated expression is found, it must be substituted by an equivalent non-negated expression before any new regions can be identified. Second column in the table shows boolean expressions from Tables 8.1, 8.2 and 8.3. The third column contains the equivalents to these expressions when they are negated. In some cases, the substitution process must be applied recursively since, for some expressions, the negated equivalent can also contain negated subexpressions.

Table 8.1: Expressions Involving Boolean Operators

	Pattern	**Regions**
1	$BExp_1 = BExp_2$	$BExp_1 = FALSE\ AND\ BExp_2 = FALSE$ $BExp_1 = TRUE\ AND\ BExp_2 = TRUE$
2	$BExp_1\ AND\ BExp_2$	$BExp_1 = TRUE\ AND\ BExp_2 = TRUE$
3	$BExp_1\ OR\ BExp_2$	$BExp_1 = FALSE\ AND\ BExp_2 = TRUE$ $BExp_1 = TRUE\ AND\ BExp_2 = FALSE$ $BExp_1 = TRUE\ AND\ BExp_2 = TRUE$
4	$BExp_1\ XOR\ BExp_2$	$BExp_1 = FALSE\ AND\ BExp_2 = TRUE$ $BExp_1 = TRUE\ AND\ BExp_2 = FALSE$
5	$BExp_1 <> BExp_2$	$BExp_1 = TRUE\ AND\ BExp_2 = FALSE$ $BExp_1 = FALSE\ AND\ BExp_2 = TRUE$
6	$Class.BAttr = TRUE$	$Class :: AllInstances() \rightarrow forAll(c\mid c.BAttr = TRUE)$
7	$Class.BAttr = FALSE$	$Class :: AllInstances() \rightarrow forAll(c\mid c.BAttr = FALSE)$

8.3.2 Analyzing OCL Expressions

Typically, real-life OCL invariants will be composed by combinations of some of the patterns described above. In the following we describe how to process some of these combined expressions, in particular, the focus is on the OCL expressions that can be characterized as $source \rightarrow operation(argument)$. For the case of more complex expressions, involving boolean (AND, OR, ...) or logical operators ($\leq$, >, ...), the process is essentially the same.

1. Find a pattern matching the whole invariant. If not found, end here.
2. Generate the new OCL expressions corresponding to the pattern matched.
3. Find a pattern matching the "source" expression.
4. If found, generate the OCL expressions corresponding to the pattern matched.
5. Repeat the process recursively over the subexpressions in the "source" expression, until no more matchings are found.

Table 8.2: Expressions Featuring Boolean Functions in the Context of a Collection

	Pattern	**Regions**
1	$col \rightarrow exists(body)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$
2	$col \rightarrow one(body)$	$col \rightarrow size() = 1$ $col \rightarrow size() > 1$
3	$col \rightarrow forAll(body)$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
4	$col \rightarrow includes(o)$	$col \rightarrow count(o) = 1$ $col \rightarrow count(o) > 1$
5	$col \rightarrow excludes(o)$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
6	$col_1 \rightarrow includesAll(col_2)$	$col_1 \rightarrow size() = col_2 \rightarrow size()$ $col_1 \rightarrow size() > col_2 \rightarrow size()$
7	$col_1 \rightarrow excludesAll(col_2)$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$
8	$col \rightarrow isEmpty()$	–
9	$col \rightarrow notEmpty()$	–

6. Find a pattern matching the "argument" expression.
7. If found, generate the OCL expressions corresponding to the pattern matched.
8. Repeat the process recursively over the subexpressions in the "argument" expression, until no more matchings are found.
9. Once the matching phase finishes, every constraint from each matching group is AND-combined with each one in the rest of the groups. This way, the final list of OCL expressions is obtained. Each of these OCL expressions characterizes a region of the input metamodel's instance space.

As an example, Fig. 8.1(b) shows another version of the metamodel describing the relationship between research teams and the papers they submit. It includes two OCL invariants. The first one states that the members of a team do not review their own papers, and the second one says that at least one of the teams must have at least one submission.

The analysis starts with the first invariant. It features a "forAll" operation matching entry 3 in Table 8.2. That entry says that the instance space can be divided in two regions. The region of models with no teams, and the one of models with any number of teams except zero. They can be characterized as:

context Team **inv** inv1: Team::AllInstances()−>isEmpty() (A1.1)
context Team **inv** inv2: Team::AllInstances()−>notEmpty() (A1.2)

Now, a pattern matching the "argument" of the "forAll" operation is searched. Entry 7 in Table 8.2 matches. Since the expression is embedded as the argument of a higher level operator, its context must be identified to build the new OCL expressions properly. By doing this, the following OCL constraints are obtained:

Table 8.3: Boolean Expressions Involving Arithmetic Operators

	Pattern	**Regions**
1	$col_1 \rightarrow size() = col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$
2	$col_1 \rightarrow size() = NUM$	$-$
3	$col_1 \rightarrow size() <> col_2 \rightarrow size()$	$col_1 \rightarrow size() > col_2 \rightarrow size()\ AND$ $\quad col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow size() < col_2 \rightarrow size()\ AND$ $\quad col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$
4	$col \rightarrow size() <> NUM\ AND$ $NUM <> 0$	$col \rightarrow size() > NUM$ $col \rightarrow notEmpty()\ AND\ col \rightarrow size() < NUM$ $col \rightarrow isEmpty()$
5	$col_1 \rightarrow size() >= col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$
6	$col \rightarrow size() >= NUM$	$col \rightarrow size() > NUM$ $col \rightarrow size() = NUM$
7	$col_1 \rightarrow size() > col_2 \rightarrow size()$	$col_2 \rightarrow isEmpty()$ $col_2 \rightarrow notEmpty()$
8	$col \rightarrow size() > NUM$	$-$
9	$col_1 \rightarrow size() <= col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$
10	$col \rightarrow size() <= NUM\ AND$ $NUM <> 0$	$col \rightarrow size() < NUM$ $col \rightarrow size() = NUM$ $col \rightarrow isEmpty()$
11	$col_1 \rightarrow size() < col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()$
12	$col \rightarrow size() < NUM$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
13	$col \rightarrow count(o) > NUM$	$col \rightarrow excluding(o) \rightarrow isEmpty()$ $col \rightarrow excluding(o) \rightarrow notEmpty()$
14	$col \rightarrow count(o) = NUM$	$col \rightarrow excluding(o) \rightarrow isEmpty()$ $col \rightarrow excluding(o) \rightarrow notEmpty()$
15	$col \rightarrow count(o) < NUM$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()\ AND$ $\quad col \rightarrow excluding(o) \rightarrow notEmpty()$ $col \rightarrow notEmpty()\ AND$ $\quad col \rightarrow excluding(o) \rightarrow isEmpty()$
16	$Class.NumAttr > NUM$	$Class :: AllInstances() \rightarrow$ $\quad forAll(c\mid c.NumAttr > NUM)$
17	$Class.NumAttr < NUM$	$Class :: AllInstances() \rightarrow$ $\quad forAll(c\mid c.NumAttr < NUM)$
18	$Class.NumAttr = NUM$	$Class :: AllInstances() \rightarrow$ $\quad forAll(c\mid c.NumAttr = NUM)$

context Team **inv** inv3: Team::AllInstances()−>forAll(t |
t.papersReviewed−>isEmpty() and t.papersSubmitted−>NotEmpty()) (A2.1)

context Team **inv** inv4: Team::AllInstances()−>forAll(t |
t.papersReviewed−>isEmpty() and t.papersSubmitted−>isEmpty()) (A2.2)

context Team **inv** inv5: Team::AllInstances()−>forAll(t |
t.papersReviewed−>NotEmpty() and t.papersSubmitted−>NotEmpty()) (A2.3)

Table 8.4: Other OCL Functions

	Pattern	**Regions**
1	$col \rightarrow select(body)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$
2	$col \rightarrow reject(body)$	$col \rightarrow forAll(NOT\ body)$ $col \rightarrow exists(body)$
3	$col \rightarrow collect(body)\ AND$ $body.oclIsTypeOf(boolean)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$
4	$col_1 \rightarrow union(col_2)$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$
5	$col_1 \rightarrow intersection(col_2)$	$col_1 = col_2$ $col_1 \rightarrow includesAll(col_2)\ AND$ $\quad col_1 \rightarrow size() > col_2 \rightarrow size()$ $col_2 \rightarrow includesAll(col_1)\ AND$ $\quad col_2 \rightarrow size() > col_1 \rightarrow size()$ $col_1 <> col_2$
6	$col \rightarrow excluding(o)$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
7	$col \rightarrow subsequence(l, u)$	$col \rightarrow size() = u - l$ $col \rightarrow size() > u - l$
8	$col \rightarrow at(n)$	$col \rightarrow size() = n$ $col \rightarrow size() > n$
9	$col \rightarrow any(body)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$

context Team **inv** inv6: Team::AllInstances()−>forAll(t |
t.papersReviewed−>NotEmpty() and t.papersSubmitted−>isEmpty()) (A2.4)

With this, the matching phase over the first invariant is over. The rest of elements in the invariant do not match any pattern. Now, the resulting two groups (A1.X and A2.X) must be combined. This produces the following list of expressions:

context Team **inv** inv7: Team::AllInstances()−>isEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>isEmpty()
and t.papersSubmitted−>NotEmpty()) (A3.1)
context Team **inv** inv8: Team::AllInstances()−>isEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>isEmpty()
and t.papersSubmitted−>isEmpty()) (A3.2)
context Team **inv** inv9: Team::AllInstances()−>isEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>NotEmpty()
and t.papersSubmitted−>NotEmpty()) (A3.3)
context Team **inv** inv10: Team::AllInstances()−>isEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>NotEmpty()
and t.papersSubmitted−>isEmpty()) (A3.4)
context Team **inv** inv11: Team::AllInstances()−>NotEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>isEmpty()
and t.papersSubmitted−>NotEmpty()) (A3.5)

Table 8.5: Boolean Expressions And Their Negated Equivalents

	Pattern	**Negated Equivalent**
1	$BExp_1 = BExp_2$	$BExp_1 <> BExp_2$
2	$BExp_1\ AND\ BExp_2$	$NOT\ BExp_1\ OR\ NOT\ BExp_2$
3	$BExp_1\ OR\ BExp_2$	$NOT\ BExp_1\ AND\ NOT\ BExp_2$
4	$BExp_1\ XOR\ BExp_2$	$BExp_1 = BExp_2$
5	$col_1 \rightarrow exists(body)$	$col_1 \rightarrow forAll(NOT\ body)$
6	$col_1 \rightarrow one(body)$	$col_1 \rightarrow select(body) \rightarrow size() <> 1$
7	$col_1 \rightarrow forAll(body)$	$col_1 \rightarrow exists(NOT\ body)$
8	$col_1 \rightarrow includes(o)$	$col_1 \rightarrow excludes(o)$
9	$col_1 \rightarrow isEmpty()$	$col_1 \rightarrow notEmpty()$
10	$col_1 \rightarrow size() = col_2 \rightarrow size()$	$col_1 \rightarrow size() <> col_2 \rightarrow size()$
11	$col_1 \rightarrow size() > col_2 \rightarrow size()$	$col_1 \rightarrow size() \leq col_2 \rightarrow size()$
12	$col_1 \rightarrow size() < col_2 \rightarrow size()$	$col_1 \rightarrow size() \geq col_2 \rightarrow size()$
13	$col \rightarrow size() \leq NUM\ AND\ NUM <> 0$	$col \rightarrow size() > NUM$
14	$col \rightarrow size() <> NUM\ AND\ NUM <> 0$	$col \rightarrow size() = NUM$
15	$col \rightarrow size() = NUM$	$(col \rightarrow size() > NUM)\ OR$ $(col \rightarrow size() < NUM)$
16	$col \rightarrow size() > NUM$	$(col \rightarrow size() = NUM)\ OR$ $(col \rightarrow size() < NUM)$
17	$col \rightarrow count(o) > NUM$	$(col \rightarrow count(o) < NUM)\ OR$ $(col \rightarrow count(o) = NUM)$
18	$col \rightarrow count(o) = NUM$	$(col \rightarrow count(o) < NUM)\ OR$ $(col \rightarrow count(o) > NUM)$
19	$col \rightarrow count(o) < NUM$	$(col \rightarrow count(o) = NUM)\ OR$ $(col \rightarrow count(o) > NUM)$
20	$Class.NumAttr > NUM$	$(Class.NumAttr < NUM)\ OR$ $(Class.NumAttr = NUM)$
21	$Class.NumAttr < NUM$	$(Class.NumAttr > NUM)\ OR$ $(Class.NumAttr = NUM)$
22	$Class.NumAttr = NUM$	$(Class.NumAttr < NUM)\ OR$ $(Class.NumAttr > NUM)$

context Team **inv** inv12: Team::AllInstances()−>NotEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>isEmpty()
and t.papersSubmitted−>isEmpty()) (A3.6)

context Team **inv** inv13: Team::AllInstances()−>NotEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>NotEmpty()
and t.papersSubmitted−>NotEmpty()) (A3.7)

context Team **inv** inv14: Team::AllInstances()−>NotEmpty() and
Team::AllInstances()−>forAll(t | t.papersReviewed−>NotEmpty()
and t.papersSubmitted−>isEmpty()) (A3.8)

With this, the analysis of the first invariant is finished. The analysis of the second invariant is analogous and yields the constraints in the group B1.X.

context Team **inv** inv15: Team::AllInstances()−>forAll(t |
t.papersSubmitted−>NotEmpty()) (B1.1)

context Team **inv** inv16: Team::AllInstances()−>exists(t |
not t.papersSubmitted−>NotEmpty()) (B1.2)

Putting all together, the analysis of the two invariants in the model of Fig. 8.1(b) yielded the groups of constraints A3.X and B1.X, respectively. Each constraint in these groups characterizes a region of the instance space. They will be the input for the test model generation phase, described in the next section.

Finally, it is important to mention that the analysis of OCL invariants is not free from inconveniences. From the example, it can be easily seen that some of the generated constraints could be simplified (for example in A3.1, if there are no "Team" instances, then there is no need to check the subexpression at the right of "and"). More importantly, some of the constraints produced in the combination stage could be inconsistent. These problems can be addressed in two different ways: adding a post-processing stage at this point to "clean" the constraints obtained, or addressing them directly during the test model creation stage (our preferred alternative, as we explain in the next section).

8.4 Partition Identification and Test Models Generation

This section details the identification of partitions and the generation of test models from the sets of constraints obtained in the previous step. Our approach provides three different alternatives depending on the effort the tester wants to invest to ensure the absence of overlapping test models.

8.4.1 Single Mode

As shown before, the analysis of one OCL invariant yields a list of new OCL expressions, each one characterizing a region of the instance space. It cannot be guaranteed though, that these regions do not overlap (that is, that they constitute a partition). Looking back at the example, this means that the regions in A3.X might overlap, and the same goes for the regions in B1.X (we have two groups here because we had analyzed two invariants). Fig. 8.4(a) and 8.4(b) illustrate the best- and worst-case scenarios when three regions are identified from the analysis of a given invariant. In the worst case, a generated test model to cover, for example, region 4, could indeed "fall into" this area, or in any of the adjacent overlapping areas labeled with a question mark (?). In this situation, when regions overlap, it is likely that generated test models do it as well.

Ensuring that a number of regions do not overlap requires additional effort, but in "Single Mode", no further effort to identify partitions is made. It simply runs the test model generator over the regions that were identified in the OCL analysis, each time passing the input metamodel (and its OCL invariants), and one of the OCL

expressions characterizing these regions. It represents a cheaper way (compared to the other alternatives) of creating test models without ensuring that they will not overlap. Running "Single Mode" over the example of Fig. 8.1(b) consists in invoking the model generator for each of the OCL expressions in A3.X and B1.X.

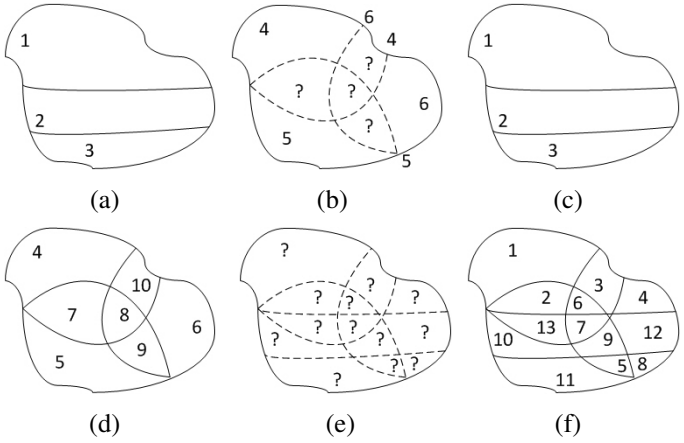


Figure 8.4: Overlapping and partitions when generating test models.

8.4.2 Multiple-Partition Mode

Given the set of OCL expressions obtained from the analysis of one OCL invariant, "Multiple-Partition Mode" produces a new set of OCL expressions that constitute a partition (i.e. do not overlap each other) of the instance space.

In general, if the analysis of one OCL invariant yields "n" regions, a partition can be derived, with a number of regions somewhere in the interval [n, 2^n - 1]. Although the exact number depends on how the original "n" regions overlap each other, justifying the lower and upper bounds is rather simple. To show this, we will focus on the particular case of $n = 3$ and refer to the OCL expressions characterizing these regions as $B_i, i = 1..3$.

The lower bound corresponds to the best-case scenario (Fig. 8.4(a)) where the original "n" regions do already constitute a partition. The upper bound corresponds to the worst-case scenario (Fig. 8.4(b)) where the "n" regions overlap each other. In this case, it is possible to derive a partition (Fig. 8.4(d)) with 7 regions, characterized by the following combinations of OCL expressions:

$D_4 = B_4\ AND\ NOT\ B_5\ AND\ NOT\ B_6$
$D_5 = B_5\ AND\ NOT\ B_4\ AND\ NOT\ B_6$
$D_6 = B_6\ AND\ NOT\ B_4\ AND\ NOT\ B_5$
$D_7 = B_4\ AND\ B_5\ AND\ NOT\ B_6$
$D_8 = B_4\ AND\ B_5\ AND\ B_6$

$D_9 = NOT\ B_4\ AND\ B_5\ AND\ B_6$
$D_{10} = B_4\ AND\ NOT\ B_5\ AND\ B_6$

That is, all the combinations of three elements (the initial number of regions) that can take two different states (to overlap, not to overlap), excepting:

$NOT\ B_5\ AND\ NOT\ B_4\ AND\ NOT\ B_6$

which is not representative of any region, since it falls out of the instance space. Generalizing for the case of "n" regions, the upper limit of 2^n - 1 is obtained.

Running "Multiple-Partition Mode" over the example of Fig. 8.1(b) consists in first, creating all the combinations of the OCL expressions in the groups A3.X and B1.X, and then invoking the model generator to process each of them. The combination of the expressions in A3.X yields a list of 255 new expressions, so only the results of combining the OCL expressions in B1.X are shown.

context Team **inv** inv272: Team::AllInstances()−>forAll(t |
t.papersSubmitted−>NotEmpty()) and Team::AllInstances()−>exists(t |
not t.papersSubmitted−>NotEmpty()) (B2.1)
context Team **inv** inv273: not Team::AllInstances()−>forAll(t |
t.papersSubmitted−>NotEmpty()) and Team::AllInstances()−>exists(t |
not t.papersSubmitted−>NotEmpty()) (B2.1)
context Team **inv** inv274: Team::AllInstances()−>forAll(t |
t.papersSubmitted−>NotEmpty()) and not Team::AllInstances()−>exists(t |
not t.papersSubmitted−>NotEmpty()) (B2.3)

8.4.3 Unique-Partition Mode

Applying "Multiple-Partition Mode" guarantees that the regions obtained for each OCL invariant do not overlap each other. However, if the input metamodel has more than one invariant, regions in the partition for one invariant might overlap regions in the partitions of the rest of invariants. "Unique-Partition Mode" guarantees that regions do not overlap each other, no matter where they come from. Therefore, in "Unique-Partition Mode" only one partition is characterized, regardless of the number of OCL invariants of the input metamodel. This can be easily seen with an example. If Fig. 8.4(c) and Fig. 8.4(d) were the partitions produced by "Multiple-Partition Mode" for two invariants, when putting together, they would overlap as shown in Fig. 8.4(e). In this scenario "Unique-Partition Mode" would yield the partition of Fig. 8.4(f).

Applying "Unique-Partition Mode" is a simple three-step process: First, "Multiple-Partition Mode" is applied over each invariant. After that, the lists of OCL expressions characterizing the regions in each partition are merged together to form one

big list. Finally "Multiple-Partition Mode" is applied over that list, to generate the final partition. Applying this mode over the example of Fig. 8.1(b) consists in merging the results of "Multiple-Partition Mode" shown before ($255 + 3 = 258$ OCL expressions) into one big list and run another iteration of "Multiple-Partition Mode" over that list. Clearly, the main problem for the practical utilization of this approach could be the combinatorial explosion in the number of regions conforming the final partition.

8.5 Creating Test Models

After having described how partitions are generated, the last step is the creation of the actual test models. Without regard of the generation mode selected, this is a pretty straightforward process. When fed with the input metamodel (and its OCL invariants) and an OCL invariant characterizing one region of the input space, the "Test Model Generator" component (Fig. 8.3) tries to build a valid instance of the input metamodel, that also satisfies this additional OCL constraint. The whole set of test models is obtained by repeating this process as many times as regions were found.

In practical terms, we use EMFtoCSP for that. At is was described in Chapter 4, this tool is capable of looking for valid instances of a given metamodel enriched or not with OCL constraints. This is especially convenient to address the issues mentioned at the end of Section 8.3. For example, when presented with an infeasible combination of constraints, EMFtoCSP can dismiss it, yielding no test model.

8.6 Implementation and Usage Scenarios

We have implemented an Eclipse-based tool that can generate test models following any of the three generation modes exposed before. It can be downloaded from http://code.google.com/a/eclipselabs.org/p/oclbbtesting/ where the user will find all the necessary information for its installation and usage.

When used in isolation, the tool produces models to cover the instance space of the transformation's input metamodel, out of the OCL invariants of that metamodel. Since graphical constraints in a model, like associations, multiplicities, etc can also be expressed in the form of OCL invariants, as detailed in [163], the tool could also be used to derive test models out of these graphical constraints.

There may be occasions though, in which it is convenient to focus only on specific sections of the input metamodel: the model transformation could only "exercise" a part of the input metamodel, or the tester could only be interested on a specific part of the transformation. In the first case, the tool could be combined

with approaches capable of identifying what the relevant sections of the input metamodel are, like for example [129]. In the second case, if the pre-conditions that trigger specific parts of the model transformation are expressed in such a way, that new OCL invariants in the context of the input metamodel can be derived, then these new invariants could be used to limit the generation of test models to those regions of the instance space triggering the sections of the model transformation that are of interest. This could be exploited even further, to allow the generation of test models aimed at satisfying different coverage criteria over the transformation [139].

Finally, the tool could also be useful to complement others that lack the ability to generate test models out of OCL invariants, or do it in a limited way.

8.7 Conclusions

The generation of test models by means of black-box approaches based on partition analysis has largely ignored the valuable information in the OCL constraints. This limits the test generation process and consequently, the degree of coverage achieved over the model transformation's input metamodel. In this chapter, we have presented a black-box test model generation approach for model transformation testing, based on a deep analysis of the OCL invariants in that input metamodel. Our method can be configured to be used at three different levels of exhaustiveness, depending on the user's needs. A tool supporting the process has been implemented, and it can be used in isolation or combined with other test model generation approaches. It can also be useful to generate test models at different degrees of coverage.

9

ATLTest: White-box Test Model Generation for ATL Model Transformations

9.1 Motivation

As it was mentioned in Chapter 7, the most popular strategy at the time of generating test models is to follow a black-box approach based on the analysis of the model transformation specification. The number of white-box approaches available is certainly limited, and even more so in the particular case of the ATL Transformation Language (ATL) [7] [1]. ATL is a popular transformation language, which means that the number of model transformations available is relatively high, especially when compared to other model transformation approaches. Therefore, the lack of specific white-box approaches to test model transformations written in this language is especially unfortunate. After all, the existence of approaches based on different strategies is encouraged to maximize the chances of uncovering errors and therefore, the effectiveness of the testing experience. Moreover, existing white-box approaches steer the generation of test models towards covering the model transformation's input metamodel instead of the model transformation internals.

In this chapter, we present a new white-box testing approach called "ATLTest", for the generation of test input models out of ATL model transformations. We have chosen ATL [7] as target transformation language due to its popularity (both in

1. http://www.eclipse.org/atl/

academia and industry). The goal is to optimize the generation of test models by maximizing the coverage of the internal structure of an ATL model transformation. However, many of the ideas presented herein could be applied to other transformation languages like QVT. The approach can be used in isolation or could be integrated with black-box testing techniques to provide a hybrid test model generation framework.

9.2 Generalities About ATL

Before getting into detail on the particularities of ATLTest, it is important to introduce a series of concepts, starting with a brief description of the ATL language.

ATL is a hybrid model transformation language, which means that combines declarative and imperative constructs. Although ATL creators encourage the utilization of a declarative style for the implementation of model transformations, imperative constructs are also provided, because in certain scenarios, implementing a model transformation only by means of declarative constructs can be cumbersome.

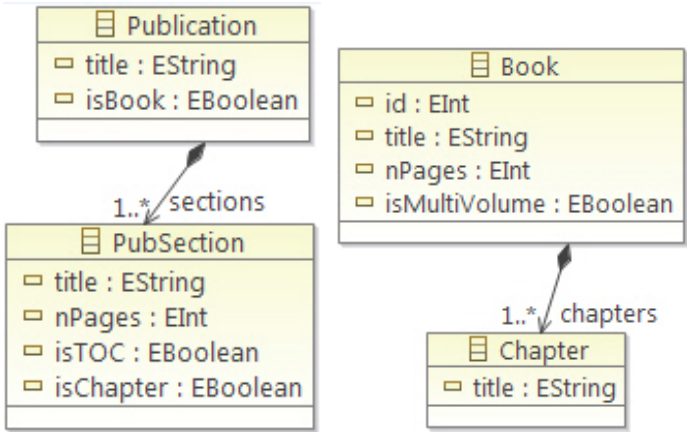
For the description of the different ATL constructs, we will take advantage of the sample model transformation definition in Fig. 9.1(c), that converts models conforming the metamodel in Fig. 9.1(a), into models conforming the metamodel in Fig. 9.1(b). In a nutshell, the model transformation contains two rules ("Publication2Book" and "PubSection2Chapter") to respectively transform "Publication" and "PubSection" input elements into "Book" and "Chapter" output elements. Those elements are only transformed if the respective flags "isBook" and "isChapter" are activated.

Model transformations in ATL are described in the context of something called "module". A module is nothing more than a collection of ATL constructs, organized in a certain way. In particular, a module contains a mandatory header section, an optional import section and a number of helpers and transformation rules.

The header section gives the name of the module and declares the source and target metamodels. This is typically done in the first two lines of code. In the model transformation definition of the example can be seen that the module is named "Publication2Book", and that the source and target metamodels are called "Publication" and "Book", respectively.

Right after the header section comes an optional import section. This section is used to import existing ATL libraries. An ATL library is a set of ATL helpers grouped in a separate file. The model transformation definition of the example does not contain an import section, but if we were interested in loading an ATL library called "strings", the import section would look like:

uses strings;



(a) (b)

```
module Publication2Book;
create OUT : Book from IN : Publication;

rule Publication2Book {
  from p: Publication!Publication (p.isBook)
  to b: Book!Book (
    title<− p.title,
    isMultiVolume<− p.sections−> select(s|
      s.isChapter)−> size() > 25 and
      p.sections−> select(s| s.isTOC)−> size() > 2,
    chapters<− p.sections−> select(s | s.isChapter),
    nPages<− p.sections−> collect(s | s.nPages)−> sum()
  )
}

rule PubSection2Chapter {
  from ps: Publication!PubSection (ps.isChapter)
  to c: Book!Chapter (
    title<−ps.title
  )
}
```

(c)

Figure 9.1: Example: (a) Source Metamodel, (b) Target Metamodel (c) Model Transformation Definition.

After the import section, it is typical to find a number of helper definitions. A helper can be seen as the ATL equivalent to a Java method. They facilitate writing ATL code that can be called from different points of an ATL transformation. ATL helpers are characterized by a name, a context, a set of parameters and a return type. The model transformation definition of the example does not contain any helpers, but if we needed a piece of code returning the publication sections larger than a specific number of pages, to be called from different points in the transformation, then we would write a helper like:

helper context Publication!Publication
def : getPubSectionsLargerThan(nPages : Integer) :
Sequence(Publication!PubSection) = self.sections−>select(s | s.nPages > nPages);

As it can be seen in the example, the helper's name is "getPubSectionsLargerThan", the context is the class "Publication" in the metamodel "Publication" (in this case both share the same name), there is only one parameter called "nPages", and the return type is an OCL sequence of "PubSection" objects. The body of the helper is an OCL query returning the list of publication sections with more pages than the minimum number requested.

Finally, after the helper definitions, it is the time of describing the most important element in an ATL model transformation definition: the transformation rules. In ATL there are three different types of rules: "matched rules", "lazy rules" and "called rules".

Matched rules are the rules most typically used in ATL, and constitute the core of the declarative nature of the language. In essence, matched rules describe for which source model elements target elements must be generated, and the way they must be initialized. The model transformation definition of the example contains two matched rules called "Publication2Book" and "PubSection2Chapter" to respectively transform "Publication" and "PubSection" input elements into "Book" and "Chapter" output elements. The "from" section in every matched rule indicates which source model element triggers the rule. Basically, every time the ATL engine finds a model element matching what it is stated in this section, the rule is triggered. It is possible to refine the triggering mechanism by adding a boolean expression between parenthesis. By doing so, only the model elements that satisfy the expression will be processed. The "to" section of the rule indicates the target model elements that will be generated. Between parenthesis is the description of how these model elements must be initialized. This description is called "bindings" in the ATL terminology. Bindings are typically expressed by a combination of OCL expressions and helper calls. To finish with the description of matched rules, it is important to mention that they may include an optional "do" section for the specification of imperative statements. It is typically used to initialize some model element features

that have not been initialized using the declarative bindings, or to modify some already initialized features.

Lazy rules are like matched rules, so the previous description applies here as well. The only difference is that, in order to execute a lazy rule, it must be explicitly called from another rule.

Finally, called rules enable the possibility of generating target model elements from imperative code. A called rule has three sections and can accept parameters. The first section is employed for the initialization of local variables, the second section (called the "to" section as in the other types of rules) indicates the target model elements that will be generated, although in this case, there is no source matched model element whose features may be used in order to initialize them. Finally, the "do" section allows for the inclusion of an imperative instruction block.

This has been a very briefly description of the ATL language to facilitate the contextualization of ATLText. A more comprehensive description of the language can be found in [7].

9.3 Coverage Criteria in Traditional White-box Testing

In Chapter 7 we mentioned that "coverage criteria" help testers to determine what elements are going to be the focus of testing, and the desired intensity of the testing efforts. Coverage criteria are not exclusive of model transformation testing, though. On the contrary, it is an expression fairly used in approaches devoted to testing software developed using more traditional means, and that, as others, has also been adopted and used when talking about model transformation testing. In what follows, we introduce very briefly, some basic concepts about some coverage criteria typically used when testing software by means of white-box techniques.

The generation of test data by means of traditional white-box testing techniques can be regarded as a 2-step process in which, typically, a control flow graph or a data flow graph is generated in the first place, out of the static analysis of the source code, and then, test data is obtained from traversing the graph a specific number of times, usually determined by some coverage criteria.

In the particular case of control flow graphs, branches in the program logic are elements typically selected as object of coverage analysis. There are a number of classical white-box coverage criteria that follow this approach, like for example, "condition coverage" (also known as "decision coverage") or "multiple-condition coverage" (also known as "multiple-decision coverage") [31]. Both focus on making sure that all branches in the program are covered, but they differ on how they exercise conditional branches where the condition is not atomic. In the case of

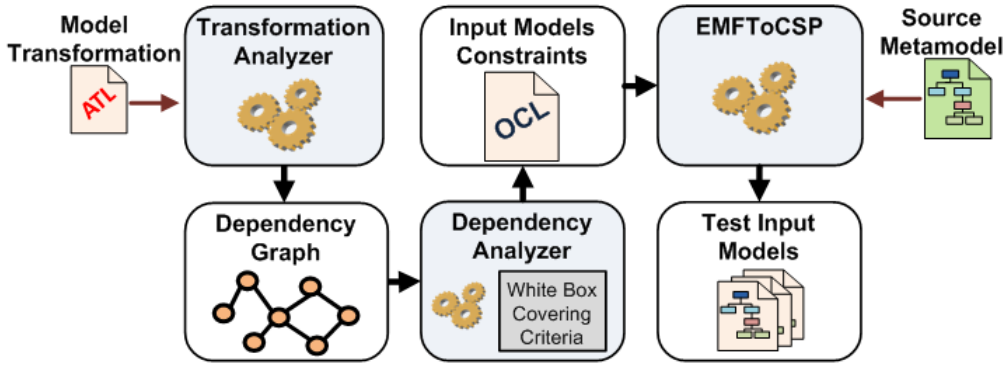


Figure 9.2: ATLTest: Overall picture

"condition coverage", complete coverage is achieved by simply ensuring that the test cases exercise each branch with all possible outcomes at least once (i.e. for a boolean branch, the test suite must include a test case where the branch evaluates to "False" and one where it evaluates to "True"). However, "multi-condition coverage" requires the test suite to include a test case for each individual combination of truth values of the different sub-expressions conforming the branch condition.

9.4 ATLTest in a Nutshell

ATLTest is a white-box test model generation approach for ATL transformations. The test model generation process in ATLTest, depicted in Fig. 9.2, consists of three separate steps. In the first one, the ATL transformation is analyzed and a graph abstracting the relevant information for the test generation phase is produced. This graph, called "dependency graph", plays the same role in ATLTest than control flow graphs or data flow graphs play in other traditional approaches, although it is substantially different in nature. For now, it suffices to say that the dependency graph represents groups of interrelated conditions expressed in the OCL. These conditions must be hold (totally or partially) by the test input models.

Once the analysis of the ATL transformation is done, the second step is to traverse the dependency graph a number of times which, as for traditional approaches, is determined by some coverage criteria. Traversing the dependency graph implies setting truth values for the different conditions in the graph and, therefore, each traversal will yield a set of constraints that symbolizes a family of relevant test models for the transformation (i.e. the constraints characterize the structure/values of possible input test models).

In the last step, the actual test models are created by computing models conforming to the source metamodel and satisfying the constraints obtained in the previous step. This computation can be performed using any of the SAT-based or CSP-based solvers available. In particular, we use EMFtoCSP to generate the input test models. In the context of model transformation testing, and given the OCL expressions

resulting from one graph traversal, EMFtoCSP will generate a solution (i.e. a test model) that satisfies both the source metamodel and these additional constraints. A single sample model suffices to characterize the family of models satisfying that particular group of constraints. Invoking EMFtoCSP after each graph traversal will result in a new test model.

In the following sections we will describe in more detail the foundations and rationale behind ATLTest.

9.5 Dependency Graph Generation

As it was mentioned before, the ATL language includes a variety of constructs (matched rules, lazy rules, helpers, etc). In most of them, OCL plays a key role. Therefore, any white-box testing approach for ATL must devote a special attention to the OCL expressions appearing in the model transformation.

In fact, OCL expressions are at the heart of the mechanism to create the dependency graph. In a nutshell, the majority of nodes and arcs are generated out of the analysis of certain OCL expressions found in the rules and helpers making up the model transformation. These nodes and arcs conform the building blocks of the dependency graph. The analysis of the rules and helpers containing those OCL expressions extends and interconnects those building blocks. The process is described in more detail in the following subsections.

9.5.1 Analysis of OCL Expressions

OCL expressions have a clear impact on the number and structure of interesting input models to use as tests for the model transformation. To ensure the coverage of the model transformation we should make sure the test models evaluate to a different result the several OCL expressions in the transformation.

As an example, consider the following expression extracted from the model transformation in Fig. 9.1(c):

p.sections−>select(s| s.isChapter)

The expression is part of a binding in the first rule, aimed at generating as many "Chapter" elements in the output model as "PubSection" elements with the flag "isChapter" set to "True" are present in the input model. Clearly, when looking at this expression we immediately think of different situations that should be tested, e.g. "What happens if there are no "PubSection" elements in the input model?" or "What happens if none of the "PubSection" elements are flagged as chapters?". Therefore, input models that test each situation (that is, an input model with no

"PubSection" elements, a model with "PubSection" elements, a model with "PubSection" elements where no one has the flag "isChapter" activated,...) should be generated by our method.

Each question above can be characterized by means of a boolean OCL expression. For the previous example, this expression could be:

context PubSection **inv** inv1: PubSection::AllInstances()−>notEmpty()

or

context PubSection **inv** inv1:
PubSection::AllInstances()−>select(s| s.ischapter)−>notEmpty()

Each expression would constitute a node in the dependency graph (meaning that generated test models should hold the condition in the node, depending on how the graph is traversed, as explained in the next section). It is also worth noting that it does not make much sense to check the second condition if the first one does not hold (we cannot create at the same time a model with no "PubSection" elements and a non-empty list of "PubSection" elements, some of them flagged as chapters), which means that the two conditions are somehow interrelated. This interrelation is the reason why we call the graph, a dependency graph. There is a dependency between the two conditions, expressed as an arc between the two nodes. Obviously, these arcs play a key role in the traversal of the graph during the test generation phase.

In the rest of this subsection we generalize this discussion to general OCL expressions. We have identified three different big groups of OCL expressions relevant to the process sketched above, namely, expressions in the context of collections (Table 9.1), iterative operations (Table 9.2) and boolean expressions (Table 9.3). Each row in the tables shows how the dependency graph is extended when finding an expression of that type in an ATL construct. The dependency graph is expressed as two ordered sets that contain the nodes (V) and the arcs (E) in the order they are created, where nodes are described with an OCL expression, and arcs are expressed as "(x,y)", "x" and "y" being the positions of the source and target nodes in the corresponding set. In this regard, "last" is used to make reference to the last position in a set, and in the case of complex OCL expressions, "$G_x(V)$" and "$G_x(E)$" make reference to the respective sets of nodes and arcs obtained from the analysis of the source expression "x". Similar for "$G_{body}(V)$" and "$G_{body}(E)$" in Table 9.2.

One important remark is that, in order to be considered for the analysis, all these OCL expressions must reference at least one element of the input metamodel, since these are the most relevant for test model generation. The identification of the OCL expressions suitable for analysis can be done by traversing the abstract syntax tree of the OCL expressions in the model transformation.

Table 9.1: Nodes and arcs generated out of OCL operations in the context of a collection

	OCL Expression	**G=(V,E)**
1	$Obj_c.[nav\|$ $nav \rightarrow notEmpty()]$	$V = \{C :: allInstances() \rightarrow$ $select(c\|c.nav \rightarrow notEmpty()) \rightarrow$ $notEmpty()\}$
2	$C :: allInstances()$ $[\rightarrow notEmpty()]$	$V = \{C :: allInstances() \rightarrow$ $notEmpty()\}$
3	$Obj_c.nav \rightarrow isEmpty()$	$V = \{C :: allInstances() \rightarrow$ $select(c\|c.nav \rightarrow isEmpty()) \rightarrow$ $notEmpty()\}$
4	$C :: allInstances() \rightarrow$ $isEmpty()$	$V = \{C :: allInstances() \rightarrow isEmpty()\}$
5	$c \rightarrow isEmpty()$	$V = \{c \rightarrow isEmpty()\} \cup G_c(V),$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
6	$c \rightarrow notEmpty()$	$V = \{c \rightarrow notEmpty()\} \cup G_c(V),$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
7	$c \rightarrow [size()\|last()\|sum()\|$ $append(o)\|flatten()\|first()\|$ $including(o)\|prepend(o)]$	$V = G_c(V),$ $E = G_c(E)$
8	$c \rightarrow [includes(o)\|count(o)\|$ $indexOf(o)\|excluding(o)]$	$V = \{c \rightarrow includes(o)\} \cup G_c(V),$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
9	$c \rightarrow excludes(o)$	$V = \{c \rightarrow excludes(o)\} \cup G_c(V),$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
10	$c \rightarrow includesAll(cl)$	$V = \{c \rightarrow includesAll(cl)\} \cup G_c(V)$ $\cup G_{cl}(V),$ $E = \{(G_c(V)[last], G_{cl}(V)[1]),$ $(G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$
11	$c \rightarrow excludesAll(cl)$	$V = \{c \rightarrow excludesAll(cl)\} \cup G_c(V)$ $\cup G_{cl}(V),$ $E = \{(G_c(V)[last], G_{cl}(V)[1]),$ $(G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$
12	$c \rightarrow union(cl)$	$V = G_c(V) \cup G_{cl}(V),$ $E = \{(G_c(V)[last], G_{cl}(V)[1])\} \cup$ $G_c(E) \cup G_{cl}(E)$
13	$c \rightarrow [insertAt(n, o)\|at(n)]$	$V = \{c \rightarrow size() \geq n\} \cup G_c(V)$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
14	$c \rightarrow subSequence(l, u)$	$V = \{c \rightarrow size() \geq u\} \cup G_c(V)$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
15	$c \rightarrow [intersection(cl)\|$ $symetricDifference(cl)]$	$V = \{c \rightarrow includesAll(cl)$ or $cl \rightarrow includesAll(c)\} \cup G_c(V) \cup G_{cl}(V),$ $E = \{(G_c(V)[last], G_{cl}(V)[1]),$ $(G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$

To finish this subsection, we show how to create nodes 3, 4, 5, 6 and 7 of the dependency graph in Fig. 9.5 by applying the information in the tables to the following expression from the example:

Table 9.2: Generation of nodes and arcs out of OCL iterative operations

	OCL Expression	**G=(V,E)**
1	$c \rightarrow exists(body)$	$V = \{c \rightarrow exists(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
2	$c \rightarrow forAll(body)$	$V = \{c \rightarrow forAll(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
3	$c \rightarrow isUnique(body)$	$V = \{c \rightarrow isUnique(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
4	$c \rightarrow one(body)$	$V = \{c \rightarrow one(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
5	$c \rightarrow [collect(body) \mid sortedBy(body)]$	$V = G_c(V)$, $E = G_c(E)$
6	$c \rightarrow [reject(body) \mid any(body) \mid select(body)]$	$V = G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1])\} \cup G_c(E) \cup G_{body}(E)$

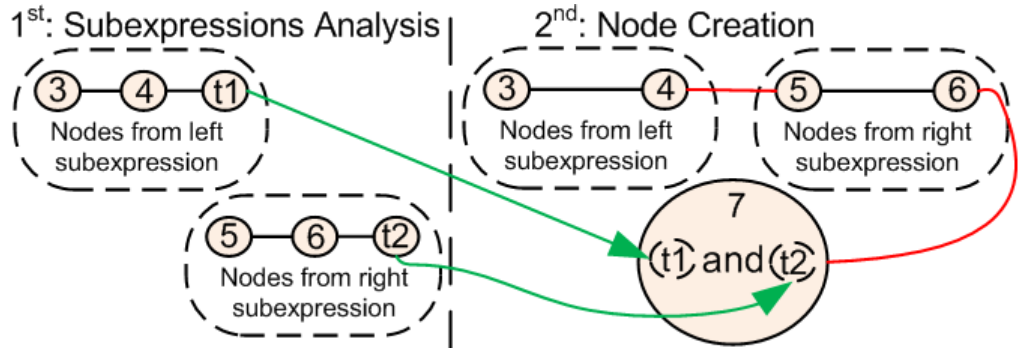


Figure 9.3: Actions to carry out when applying entry 10 in Table 9.3 to the example

isMultiVolume<−p.sections−>select(s| s.isChapter)−>size() > 25
and p.sections−>select(s| s.isTOC)−>size() > 2 (exp1)

To begin with, the OCL expression at the right of "<−", matches entry 10 in Table 9.3 using "and" as "Op". According to this entry, the 2-step process depicted in Fig. 9.3 must be carried out. That is, subexpressions at the left and right of "and" must be analyzed, thus yielding several nodes and arcs, and then some of those nodes must be merged. Finally all the nodes are interconnected.

The expression on the left side of (exp 1) is

p.sections−>select(s| s.isChapter)−>size() > 25 (exp2)

that matches entry 11 in Table 9.3 where "CompOp" is ">" and 'LitValue" is "25". Fig. 9.4 illustrates the process to be carried out when instructions in this entry are followed. The subexpression on the left side is analyzed in the first place, this way yielding nodes 3 and 4, and then, the node "t1' is created.

Table 9.3: Generation of nodes and arcs out of boolean OCL operations

	Boolean OCL Expression	**G=(V,E)**
1	$[True\|False]$	$V = \varnothing, E = \varnothing$
2	$[not]Obj_A.boolAttr$	$V = \{A :: allInstances() \rightarrow select(a\|[not]a.boolAttr) \rightarrow notEmpty()\}$
3	$Obj_A.[attr\|nav].oclIsUndefined()$	$V = \{A :: allInstances() \rightarrow select(a\|a.[attr\|nav].oclIsUndefined()) \rightarrow notEmpty()\}$
4	$expr.oclIsUndefined()$	$V = \{expr \rightarrow oclIsUndefined()\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
5	$expr.oclIsKindOf(t)$	$V = \{expr \rightarrow oclIsKindOf(t)\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
6	$expr.oclIsTypeOf(t)$	$V = \{expr \rightarrow oclIsTypeOf(t)\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
7	$Obj_A.attr\ CompOp\ LitValue$	$V = \{A :: allInstances() \rightarrow select(a\|a.attr\ CompOp\ LitValue) \rightarrow notEmpty()\}$
8	$Obj_A.attr\ Op\ Obj_B.attr$	$V = \{A :: allInstances() \rightarrow select(a\|B :: allInstances() \rightarrow exists(b\|a.attr\ CompOp\ b.attr)) \rightarrow notEmpty()\}$
9	$Obj_A.attr\ Op\ Obj_B.attr\ Op\ ...\ Op\ Obj_N.attr$	$V = \{A :: allInstances() \rightarrow select(a\|B :: allInstances() \rightarrow exists(b\|... \rightarrow exists(n\|a.attr\ Op\ b.attr\ Op\ ...\ Op\ n.attr)...)) \rightarrow notEmpty()\}$
10	$expr_1\ Op\ expr_2$	$V = \{G_{expr_1}(V)[last]\ Op\ G_{expr_2}(V)[last]\} \cup \{G_{expr_1}(V)[1], ..., G_{expr_1}(V)[last-1]\} \cup \{G_{expr_2}(V)[1], ..., G_{expr_2}(V)[last-1]\}$, $E = \{(G_{expr_1}(V)[last-1], G_{expr_2}(V)[1]), (G_{expr_2}(V)[last], 1)\} \cup \{G_{expr_1}(E)[1], ..., G_{expr_1}(E)[last-1]\} \cup \{G_{expr_2}(E)[1], ..., G_{expr_2}(E)[last-1]\}$
11	$expr\ CompOp\ LitValue$	$V = \{expr\ CompOp\ LitValue\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$

Now let's see in detail how nodes 3 and 4 are generated. The expression on the left of (exp 2) is

p.sections−>select(s| s.isChapter)−>size() (exp3)

that matches entry 7 in Table 9.1. According to this entry, it is necessary to analyze the source collection of (exp3), that is:

p.sections−>select(s| s.isChapter) (exp4)

It matches entry 6 in Table 9.2. This entry indicates that (exp4) has the form $c \rightarrow select(body)$ and therefore "c" and "body" expressions must be analyzed. In (exp4), "c" is $p.sections$ and "body" is $s.isChapter$. They match respectively entry 1 in

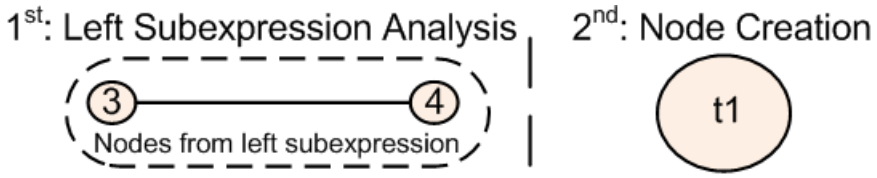


Figure 9.4: Actions to carry out when applying entry 11 in Table 9.3 to the example

Table 9.1 and entry 2 in Table 9.3. This way, we finally obtain nodes 3 and 4 that can be seen in Fig. 9.5. It is important to remember that the creation of nodes 3 and 4 is just the first step in the analysis of (exp2), as exposed in Fig. 9.4. Now it is time to complete the analysis of this expression by creating the node "t1". This node is made up by the following OCL expression:

$$p.sections->select(s|\ s.isChapter)->size() > 25 \qquad (t1)$$

It is the time to remember that the analysis of (exp2) is just the analysis of the left subexpression of (exp1). As can be seen in Fig. 9.3 the analysis of (exp1) continues with the analysis of its right subexpression. We omit a detailed description of this analysis, though, since it is very similar to the one just described. It suffices to say that the analysis of the right subexpression of (exp1) yields nodes 5 and 6 that can be seen in Fig. 9.5, as well as node "t2", made up by the following OCL expression:

$$p.sections->select(s|\ s.isTOC)->size() > 2 \qquad (t2)$$

Finally, applying last step shown in Fig. 9.3, node 7 is created out of the union of nodes "t1" and "t2", expressed in terms of the "allInstances()" operator, and the different nodes created during the process are interconnected. The final result can be seen in Fig. 9.5.

The analysis of the rest of OCL expressions in the sample model transformation can be carried out in the same way.

9.5.2 Analysis of Rules and Helpers

As we have seen, the analysis of OCL expressions yields the building blocks of the dependency graph. In this subsection we cover the analysis of rules and helpers, coarse-grained elements of ATL transformations.

The analysis of a declarative rule focuses on the "from" section of the rule, that indicates the conditions that trigger the rule, the "to" section of the rule, that describes how elements of the target model are created, and the optional "do" section of the rule, used to enable the specification of imperative statements.

The analysis of the "from" section produces a node with the expression $in\_type :: allInstances() \rightarrow notEmpty()$, where "in_type" refers to the model element that will be matched by the rule. Optionally, this section can include a boolean OCL expression, as a filter to limit the "in_type" elements that can trigger the rule. When

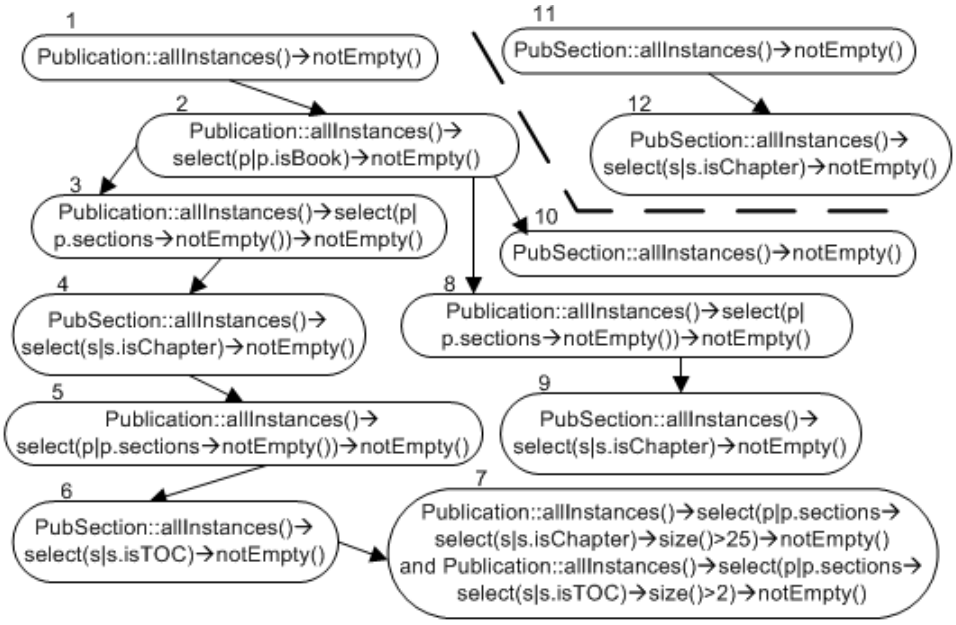


Figure 9.5: Dependency graph of the example, made up by two connected components

present, this filter is analyzed according to the instructions of Subsection 9.5.1 and, in this case, the node created in the first place is connected to the first node rendered by the filter analysis.

Returning to the running example, the analysis of the "from" section of the rule "Publication2Book", that includes the condition $p.isBook$, produces nodes 1 and 2 in Fig. 9.5. Analogously, the "from" section of the rule "PubSection2Chapter" generates nodes 11 and 12 that made up the second connected component of the dependency graph.

As it was mentioned before, the "to" section of a declarative rule is, essentially, a collection of bindings describing how elements of the target metamodel are created. Each binding has the form $feature\text{-}name \leftarrow exp$, being "exp" an OCL expression. The result of analyzing this section is a number of interconnected nodes, obtained from the analysis of each "exp" element as explained in Subsection 9.5.1. Finally, the first node in each of the groups of nodes rendered is connected to the last node in the group of nodes obtained from the analysis of the "from" section of the rule.

The "do" section of a declarative rule is analyzed by looking for OCL expressions suitable for analysis. When found, those expressions are analyzed according to the directions of Subsection 9.5.1. This approach is also applied at the time of analyzing called rules.

To finish the description of the dependency graph generation process, one word about ATL helpers. As it was mentioned before, helpers can be viewed as the ATL equivalent to methods and can be called from different points in an ATL transformation. Each helper has a body, specified as an OCL expression. If during the analysis of the elements described above and in Subsection 9.5.1, a call to a helper is found,

then its body is analyzed like any other OCL expression and the rendered nodes are included as resulting from the analysis of the element where the call was found.

One last remark that is worth mentioning is that depending on the complexity of the ATL transformation under analysis (number of rules, presence of imperative sections, etc.), the resulting dependency graph can be made up by more than one connected component.

9.6 Test Input Models Generation

Once the dependency graph is created, the next step consists in traversing it a number of times, each time determining the set of constraints a new test model must fulfill. The process is directed by a coverage criterion, which eventually determines the number of traversals, and consequently, the number of test models to be generated.

Coverage criteria mentioned before such as "condition coverage" or "multiple-condition coverage" can be easily adapted to our approach. Since in the dependency graph each node contains a boolean expression, condition coverage and multi-condition coverage can be applied by considering each node as a branch, with the particularity that every time the condition in the node evaluates to "False" the traversal of the actual connected component ends and goes on with the next one. In other case, a neighbor node is visited and the traversal continues.

This way, the application of the two coverage criteria consists in traversing the dependency graph a number of times, each time assigning either different output values to each OCL expression (condition coverage), or different combinations of truth values to each component of a complex OCL expression (multi-condition coverage). After "n" traversals, "n" sets of constraints to characterize "n" test cases will have been obtained.

Eventually, once the sets of constraints have been obtained, the execution of EMFtoCSP over each set will yield the set of input models to test the model transformation [2].

Retaking our example, we are going to show what the results of one traversal of the graph shown in Fig. 9.5 would be when applying each coverage criterion. Let's suppose that the sequence of truth values assigned to the nodes of the first connected component is <1,True>, <2,True>, <3,True>, <4,True>, <5,True>, <6,True>, and then, in the case of "condition coverage" node 7 is set to <7,True>, and in the case of "multi-condition coverage" is set to <7,(False,True)>. In the second connected component, the expressions will be set as <11,True>, <12,True>, for both criteria.

2. Some assignments can cause contradictory sets of OCL expressions (e.g. if the same subexpressions are used in two connected components and they are assigned different truth values in the same iteration). In those situations, EMFtoCSP will yield no test model.

Applying "condition coverage", the constraints obtained are:

context Publication **inv** inv1:
Publication::AllInstances()−>notEmpty() = true

context Publication **inv** inv2:
Publication::AllInstances()−>select(p| p.isBook)−>notEmpty() = true

context Publication **inv** inv3:
Publication::AllInstances()−>select(p|
p.sections−>notEmpty())−>notEmpty() = true

context PubSection **inv** inv4:
PubSection::AllInstances()−>select(s|s.isChapter)−>notEmpty() = true

context Publication **inv** inv5:
Publication::AllInstances()−>select(p|
p.sections−>notEmpty())−>notEmpty() = true

context PubSection **inv** inv6:
PubSection::AllInstances()−>select(s| s.isTOC)−>notEmpty() = true

context PubSection **inv** inv7:
Publication::AllInstances()−>select(p| p.sections−>select(s|
s.isChapter)−>size() > 25)−>notEmpty() and
Publication::AllInstances()−>select(p| p.sections−>select(s|
s.isTOC)−>size() > 2)−>notEmpty() = true

context PubSection **inv** inv8:
PubSection::AllInstances()−>notEmpty() = true

context PubSection **inv** inv9:
PubSection::AllInstances()−>select(s|s.isChapter)−>notEmpty() = true

Running EMFtoCSP over the input metamodel constrained with the expressions above yields the model that can be seen in Fig. 9.6(a).

For "multi-condition coverage", only the expression of node 7 changes:

context PubSection **inv** inv7:
Publication::AllInstances()−>select(p| p.sections−>select(s|
s.isChapter)−>size() > 25)−>notEmpty() = false and

Publication::allInstances()−>select(p| p.sections−>select(s| s.isTOC)−>size() > 2)−>notEmpty() = true

Running again EMFtoCSP, we obtain the model of Fig. 9.6(b).

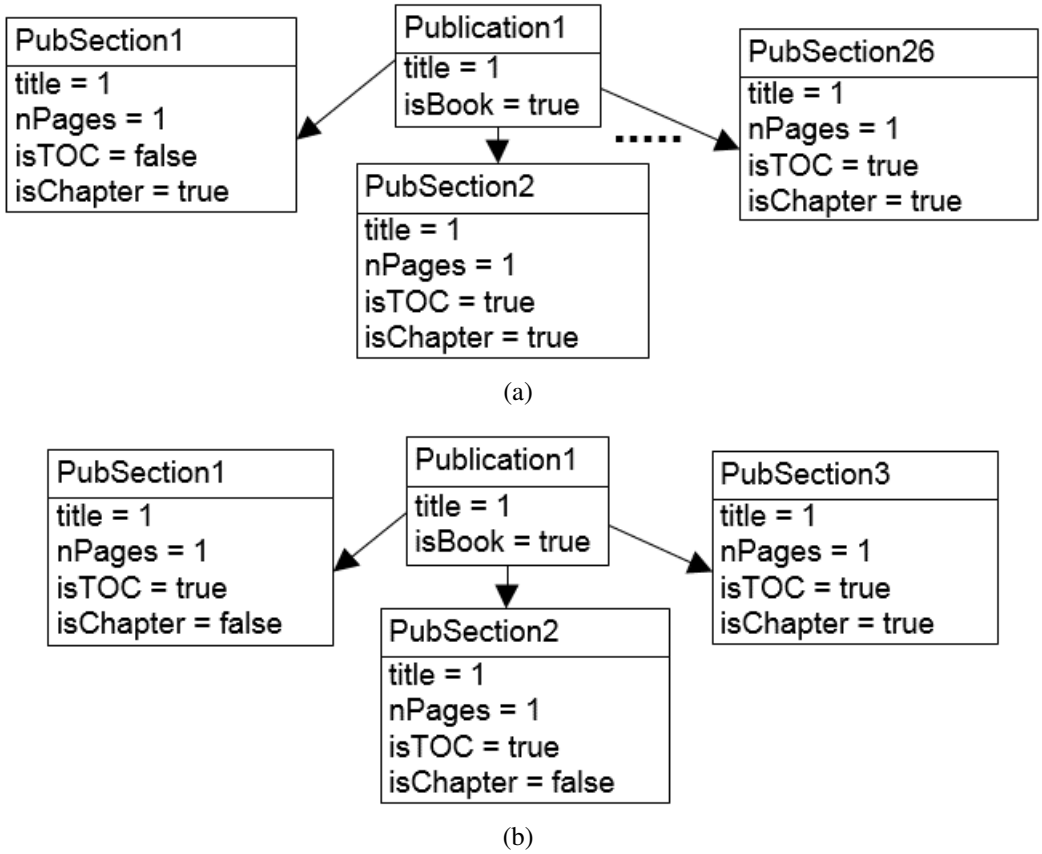


Figure 9.6: Results of the example

9.7 Conclusions

We have presented ATLTest, a white-box testing approach for the generation of test input models for ATL transformations. Our approach tries to optimize the effectiveness of the generated tests by maximizing the coverage of the internal structure of the model transformation under analysis. ATLTest could be either combined with black-box testing techniques to create mixed test generation approaches or used in isolation, specially in those scenarios that are not suitable for the application of black-box testing techniques. In ATLTest, each test model is characterized by a set of OCL expressions. Sample test models satisfying the OCL constraints are created automatically using the EMFtoCSP tool.

IV

Conclusions and Future Research

10

Conclusions and Future Research

10.1 Conclusions

This thesis analyzes the landscape of approaches devoted to the verification of static models with the twofold objective of presenting contributions to improve their efficiency, and discovering new mechanisms that take advantage of verification approaches at the time of testing model transformations.

The analysis of the state of the art (Chapter 3) has revealed that there is no a universal definition of what "model correctness" means and therefore, it is typical to use this expression to make reference to the ability of the model under analysis to satisfy one or more correctness properties. Regarding the verification process, it is common to represent the problem using some kind of formalism, which is then exploited with the help of a specialized tool. In spite of that, existing approaches follow different strategies depending on the degree of support given to OCL constraints. Regarding this, bounded verification approaches consisting in limiting the search space where to look for a solution of the problem are becoming more and more popular.

Unfortunately, the number of existing verification approaches is rather small, and some of them are not even supported by the presence of a tool. An example of this is that the Eclipse platform, which is probably the most popular software development platform at the time of promoting the utilization of model-based technologies, does not contain an official project devoted to the verification of models. In order to alleviate this, we propose a bounded verification approach called EMFtoCSP (Chapter 4), capable of verifying EMF models enriched with OCL constraints, by means of constraint programming.

The major inconvenience of existing verification tools is probably their efficiency. These tools tend to behave badly when they are used to verify large models or real-world examples. A possible way of improving this is by verifying models in an incremental fashion (Chapter 5). If every time a model is modified, only certain parts of the model need to be verified again, then the size of the models to be analyzed by verification tools could be reduced, which might improve performance. Additionally, if the verification tool employed follows a bounded verification approach, search space boundaries could also be tightened (Chapter 6) to reduce the area where to look for a solution.

Before analyzing how to take advantage of verification approaches to conduct model transformation testing, it is necessary to identify what are the major challenges. The analysis of the state of the art (Chapter 7) has shown that model transformation testing is, in essence, similar to testing traditional software. When dynamic techniques are used, the challenges are the generation of input test data (known as test models) and the construction of oracles. In the particular case of test model generation, two strategies prevail: the black-box strategy based on the analysis of the model transformation specification, and the white-box strategy, based on the analysis of model transformation internals. Unfortunately, the number of existing approaches is really small, especially in the case of approaches following a white-box strategy.

In order to alleviate this, two test model generation techniques has been proposed. The first one, following a black-box approach (Chapter 8), exploits the presence of OCL constraints on static models to build partitions of the model transformation's input metamodel instance space, which are then exploited by means of EMFtoCSP to generate test models. In the second approach (Chapter 9), OCL expressions present in ATL model transformations are analyzed to, again with the help of EMFtoCSP, generate test models.

10.2 Future Work

We would not like to finish this thesis without briefly describing some ideas to expand the work presented here. In particular, the following subsection presents some general ideas that could be worth being explored to try to improve existing verification tools. After that, the focus is on ideas specifically devoted to try to expand the methods and techniques presented here.

10.2.1 General Ideas for the Improvement of Verification Tools

Existing verification tools work by translating the verification problem into some kind of formalism that is then exploited with the help of a specialized tool. When

the verification problem is found to be not satisfiable, some of these tools (SAT solvers, SMT solvers, CSP solvers) are capable of yielding additional information in the form of unsatisfiable cores identifying contradictory constraints that can be analyzed to debug the problem. We think that the analysis of this information could also be helpful to provide users with a more meaningful feedback on which are the conflicting regions of the model under analysis.

Another drawback of this kind of tools is that the reasoning process can take a huge amount of time, which is especially true when they have to face the exploration of large search spaces. In these cases, it would be interesting to analyze whether it is possible to extract useful information for the user when the verification process is executed only for a limited period of time. Information like for example, the probability for a given model to be valid or not valid, depending on the size of the search space explored could be valuable for certain users. This could also be an improvement when compared to the current situation, where if the verification process is stopped after being running for some time, the only information the user can infer is that no valid instance was found.

10.2.2 EMFtoCSP

Apart from being our tool of choice to try to develop the ideas exposed in the previous subsection, we have some other ideas involving EMFtoCSP. In particular, we are interested in exploring the translation of models into CSPs on infinite domains. In these cases, constraint solvers allow incomplete search [120] where termination is not guaranteed and heuristics are needed to guide the search process. Additionally, we would also like to explore how to improve the efficiency of the generated CSP by refining our translation process. Finally, although not exactly related to EMFtoCSP, we are also interested in investigating the utilization of SMT solvers for static model verification.

10.2.3 Incremental Verification of Models

Admittedly, our approach for the incremental verification of models is rather conservative. However, experimental results have also shown that the time needed to calculate the submodel that must be re-verified is negligible, compared to the time spent during the verification process. As future work, we plan to further analyze some scenarios to reduce even more the size of this submodel. In particular, we know that in certain cases, modifying OCL invariants and generalizations do not impact the satisfiability of the model. It is also our intention to study the integration of this approach with model editing tools. At this moment, the success of model verification tools is far from being great. We think this could change if we provide

model designers with more comfortable ways of using verification tools. In this regard, we think that incremental verification approaches can be of great help.

10.2.4 Adjusting Search Space Boundaries

Regarding the calculation of boundaries for bounded verification tools, we plan to investigate heuristics regarding the best order for selecting bounds, that is, one that reduces the number of choices and maximizes the amount of information that can be inferred automatically by the proposed method. We also intend to investigate how to reverse this approach, for example, by broadening (instead of tightening) user provided bounds which are too strict to find a counterexample. Finally, we will consider the utilization of heuristics to suggest promising values for bounded domains based on the constraints of the model.

10.2.5 Model Transformation Testing: Black-box Approaches

In relation to the utilization of black-box test generation approaches for model transformation testing, we would like to improve the way OCL expressions characterizing regions of the instance space are combined. The objective here is to reduce the number of spurious or infeasible combinations produced. Additionally, and regarding the OCL expressions characterizing regions of the instance space themselves, in the majority of cases, they are generated exploiting the presence of collections of elements in the original invariant, by deriving new expressions where these collections are forced to be either empty or not empty. In the future, we would like to use more complex criteria to try to identify other groups of expressions also characterizing regions of the instance space.

10.2.6 Model Transformation Testing: White-box Approaches

Finally, when it comes to white-box test model generation, we plan to extend our to approach to cover other transformation languages like QVT. We would also like to study complexity metrics like cyclomatic complexity [164] to establish a limit on the number of test models that need to be generated, something that can be specially useful when testing large transformations.

A

Résumé en Françaís

A.1 Introduction et Objectifs

L'Ingénierie Dirigée par les Modèles (IDM) est une approche populaire pour le développement logiciel qui favorise l'utilisation de modèles au sein des processus de développement. Dans un processus de développement logiciel base sur l'IDM, le logiciel est développé en créant des modèles qui sont transformés successivement en d'autres modèles et éventuellement en code source. Quand l'IDM est utilisée pour le développement de logiciels complexes, la complexité des modèles et des transformations de modèles augmente, risquant d'affecter la fiabilité du processus de développement logiciel ainsi que le logiciel en résultant. Traditionnellement, la fiabilité des logiciels est assurée au moyen d'approches pour la vérification de logiciels, basées sur l'utilisation de techniques pour l'analyse formelle de systèmes et d'approches pour le test de logiciels. Pour assurer la fiabilité du processus IDM de développement logiciel, ces techniques ont en quelque sorte été adaptées pour essayer de s'assurer la correction des modèles et des transformations de modèles associées. L'objectif de cette thèse est de fournir de nouveaux mécanismes améliorant les approches existantes pour la vérification de modèles statiques, et d'analyser comment ces approches peuvent s'avérer utiles lors du test des transformations de modèles.

A.2 Contributions de cette Thèse

Les contributions de cette thèse sont les suivantes:

- Un mécanisme basé sur la programmation par contraintes appelé EMFtoCSP, assurant la validité des modèles statiques.
- Deux mécanismes améliorant la performance des méthodes de vérification de modèles.
- Un premier mécanisme de génération de modèles de test pour transformations de modèles, basé sur l'analyse de la structure interne des transformations.
- Un second mécanisme de génération de modèles de test pour transformations de modèles, basé sur l'analyse de la spécification des transformations.

A.3 Vérification de Modèles Statiques

A.3.1 Etat de L'Art

La vérification formelle de modèles statiques fait référence aux méthodes utilisées pour prouver la validité des modèles statiques au moyen de l'utilisation des méthodes formelles et des techniques pour l'analyse formelle de systèmes. Cependant, il n'y pas de définition universelle de la "validité des modèles". Au contraire, il y a plusieurs façons pour un modèle d'être considéré comme correct. Pour cette raison, il est courant de se référer à la "validité des modèles" comme la capacité du modèle analysé à satisfaire une ou plusieurs propriétés de validité. Ces propriétés indiquent certaines caractéristiques que le modèle doit satisfaire pour être considéré comme correct. La propriété la plus importante est appelé "satisfiabilité". Un modèle est considéré satisfiable quand il est possible de l'instancier.

La vérification de modèles statiques peut être divisée en deux étapes différentes. Dans la première se déroule la formalisation, c'est-à-dire que le modèle avec les propriétés à vérifier sont représentés dans le formalisme choisi. A partir de cette représentation formelle, la deuxième étape peut commencer. Elle consiste à raisonner sur cette représentation, habituellement avec l'aide de certains outils spécialisés, pour voir si les propriétés sont satisfaites ou non. L'exhaustivité et le degré d'automatisation de ce raisonnement dépendent fortement du degré de support fourni avec le langage de contraintes OCL. La raison à cela est que le soutien complet à OCL conduit à des problèmes d'indécidabilité, comme indiqué dans [44]. Dans ce scénario, les approches de vérification qui sont complètes sont aussi celles qui sont interactives [70, 87] (elles ont besoin de l'aide de l'utilisateur pour orienter le processus de vérification). Cela peut être problématique car cela requiert des utilisateurs experts dans les formalismes utilisés. Pour cette raison, la majorité des approches qui soutiennent complètement OCL [45, 73, 75] sont automatisées et assurent la terminaison, mais ceci au détriment de leur exhaustivité. Cela se fait en suivant une approche de vérification bornée dans laquelle les utilisateurs doivent typiquement configurer d'avance certains paramètres pour conduire le processus de

raisonnement. Mais une fois qu'il est lancé, l'intervention de l'utilisateur n'est plus nécessaire. En particulier, il est habituel que ces approches donnent aux utilisateurs la responsabilité de fixer les limites de l'espace de recherche où trouver une solution du problème. Dans ce scénario, les résultats sont concluants seulement si le modèle s'avère correct. D'autre part, les approches qui soutiennent seulement une partie d'OCL sont automatisées et exhaustives [49, 66] parce qu'elles ne sont pas affectées par les problèmes d'indécidabilité indiqués dans [44]. Finalement il est important de noter que, dans la majorité des cas où le modèle s'avère correct, ces outils donnent une instance valide de ce modèle comme preuve de validité.

A.3.2 Défis et Domaines D'Amélioration

Le premier problème important détecté est l'absence d'une terminologie précise et rigoureuse partagée entre toutes les approches de vérification étudiées. Une des conséquences de cela est la difficulté de contextualiser les approches dans ce domaine, surtout en comparaison avec les approches dans d'autres domaines similaires. Clairement, une définition non-ambigüe de tous les termes liés au concept de la vérification de modèles (comme validation, consistance, etc.) est nécessaire. L'absence d'une terminologie précise et rigoureuse affecte aussi la manière dont les propriétés de validité sont appelées et définies. à notre avis, cette absence d'homogénéité pour appeler et définir avec précision les propriétés de validité pourrait être améliorée par la création d'un catalogue de propriétés dans lequel on peut trouver la liste des propriétés qui peuvent être analysées lors de la vérification de modèles statiques. à notre connaissance, aucun travail n'a été réalisé jusqu'à présent dans ce sens.

Une autre constatation est la difficulté d'évaluer et de comparer la couverture et la performance des outils de vérification existants. Nous pensons qu'une voie possible pour améliorer la situation actuelle serait la création de benchmarks, comme cela est généralement fait dans d'autres communautés.

Une autre constatation importante concerne l'adéquation des outils de vérification existants. À notre avis, un outil de vérification, pour être efficace et largement adopté, doit présenter au moins quatre caractéristiques importantes. Premièrement, il doit cacher toutes les complexités découlant de l'utilisation des méthodes formelles, au point de rendre leur présence transparente pour l'utilisateur final. Deuxièmement, il doit s'intégrer parfaitement avec le reste des outils utilisés par l'utilisateur. Troisièmement, il doit fournir à l'utilisateur des retours significatifs. Et, plus important encore, quatrièmement il devrait être raisonnablement efficace, i.e. ne pas faire attendre les utilisateurs trop longtemps lors de la vérification de grands modèles ou de modèles du monde réel. Nous pensons que ces aspects sont, du point de vue de l'utilisateur, plus important que d'autres aspects plus formels tels

que l'exhaustivité des résultats. Malheureusement, aucun des outils de vérification analysés dans cette étude ne remplit suffisamment toutes ces exigences. En général ces outils ne s'intègrent pas bien, et ont été conçus pour effectuer le processus de vérification séparément du reste des tâches caractérisant le travail d'un créateur de modèles. Lorsqu'il s'agit de masquer les méthodes formelles employées, la situation est meilleure. C'est surtout le cas des approches de vérification bornées, mais le fait d'avoir à régler manuellement les limites de l'espace de recherche peut être un problème lors de la vérification de grands modèles. Les retours fournis peuvent être considéré comme acceptable lorsque le modèle en cours d'analyse s'avère valide mais sont insuffisants dans l'autre cas, la majorité des outils ne produisant aucun retour détaillé si le modèle ne s'avère pas valide (à notre connaissance, seul [51] fournit quelques conseils pour aider les utilisateurs dans ce cas). Finalement, l'efficacité est un problème sérieux. En général, les outils se comportent bien lorsqu'ils traitent des exemples simples ou des modèles de taille réduite. Mais la performance diminue considérablement quand ils sont utilisés pour vérifier de grands modèles ou des exemples du monde réel.

A.3.3 EMFtoCSP: Vérification des Modèles Statiques dans Eclipse

EMFtoCSP est un outil pour la vérification de modèles statiques qui peuvent inclure ou non des contraintes OCL diverses et variées. L'outil est une évolution de l'outil UMLtoCSP [45], et a pour but de couvrir l'espace laissé par l'absence de projets voués à la vérification de modèles statiques dans la plateforme Eclipse. Dans EMFtoCSP, le modèle d'entrée avec ses contraintes ainsi que les propriétés à vérifier sont convertis en un problème de satisfaction de contraintes ou CSP. Ensuite, un solveur de contraintes aussi appelé Eclipse est utilisé pour déterminer si une solution pour le CSP existe ou non. Le CSP est construit d'une manière telle qu'il a une solution si et seulement si le modèle ainsi que les contraintes satisfont les propriétés de validité choisies. Si une solution est trouvée, l'outil fournit une instance valide du modèle d'entrée comme preuve. Dans sa version actuelle, EMFtoCSP peut vérifier les propriétés de validité suivantes:

- Satisfiabilité forte: il est possible de trouver une instance valide du modèle avec une population non-vide pour toutes ses classes et associations.
- Satisfiabilité faible: il est possible de trouver une instance valide du modèle avec une population non-vide pour une de ses classes.
- Absence de contraintes englobées: étant donné un modèle avec deux contraintes OCL C1 et C2, il est possible de trouver une instanciation finie où C1 est satisfaite et pas C2.
- Absence de contraintes redondantes: étant donné un modèle avec deux contraintes OCL C1 et C2, il est possible de trouver une instanciation finie où

seule une contrainte est satisfaite.

Comparé à UMLtoCSP, EMFtoCSP comprend (entre autres améliorations) la capacité de vérifier une plus grande variété de modèles ainsi qu'une version revisité du mécanisme de génération de CSP. En particulier, l'outil peut également être utilisé pour vérifier des langages de modélisation spécifiques à des domaines donnés (DSMLs). En ce qui concerne le mécanisme de génération de CSP, grâce au travail de Büttner et al. [107], EMFtoCSP peut également analyser les expressions OCL incluant des opérations générales sur chaînes de caractères.

A.3.4 Vérification Incrémentale de Modèles

Une première étape nécessaire pour augmenter le taux d'adoption des outils de vérification est d'améliorer la performance du processus de vérification. La première variante que nous proposons pour réaliser cet objectif consiste à vérifier le modèle de façon progressive. Ainsi, chaque fois que le modèle est modifié, seules les parties pertinentes sont vérifiées à nouveau. Les approches traditionnelles vérifient le modèle complet, sans tenir compte de la nature des modifications effectuées. L'approche incrémentale proposée est également basée sur l'utilisation des outils de vérification utilisés par les approches traditionnelles (comme EMFtoCSP). Mais, au lieu de vérifier le modèle complet, il analyse d'abord les modifications faites sur le modèle, supprime celles qui n'ont pas d'impact sur sa validité et utilise le reste pour calculer la partie du modèle qui a vraiment besoin d'être vérifiée à nouveau. De cette façon, la taille du modèle qui doit réellement être vérifié à nouveau peut être réduite, contribuant ainsi à améliorer la performance du processus de vérification.

Cependant, il est important de préciser qu'une approche de vérification incrémentale est très dépendante de la propriété de validité qui est à vérifier. En fonction de la propriété de validité à analyser, quelques modifications de modèle peuvent ou non avoir un impact sur la validité du modèle. Notre approche se concentre sur la propriété appelée "satisfiabilité" et suit une stratégie assez conservatrice au moment de l'analyse des modifications sur le modèle. Diverses modifications ont été identifiées comme non-problématiques pour la validité du modèle et, par conséquent, ne sont pas à vérifier de nouveau (c'est le cas de l'ajout d'une classe, de la suppression d'une classe, d'associations binaires, et de l'ajout d'une association binaire avec des cardinalités 0 ou 1). Le reste des modifications sont considérées comme des scénarios complexes, où l'impact sur la validité du modèle n'est pas clair. Par conséquent, dans ces cas, les parties du modèle affecté par les modifications sont vérifiées à nouveau au moyen d'EMFtoCSP.

A.3.5 Limiter L'Espace de Recherche des Méthodes de Vérification Bornées

Une des stratégies les plus populaires de la vérification de modèles statiques est de suivre une approche de vérification bornée consistant à limiter l'espace de recherche où trouver une solution du problème de vérification. Malheureusement, la définition des limites de l'espace de recherche s'est avérée être un facteur limitant parce que les outils existants définissent des valeurs inadéquates par défaut, ou forcent les utilisateurs à définir ces valeurs manuellement. La définition manuelle de ces limites est un gros problème parce que cela nécessite la spécification de limites supérieures et inférieures pour la population de chaque classe et association dans le modèle, ainsi que les domaines pour chaque attribut. Lorsqu'il s'agit de grands modèles, cela se traduit par la définition des valeurs limites pour un ensemble de centaines d'éléments de modèle ou même plus. En outre, il n'y a pas de garantie que ce processus ne doive pas être répété plusieurs fois afin d'affiner l'espace de recherche résultant. Bien sûr, il y a toujours le risque que l'utilisateur fasse une erreur quand elle/il définit les domaines, lui laissant penser que le modèle n'est pas valide alors qu'il s'agit juste d'une erreur dans la sélection des limites.

Notre seconde approche pour améliorer la performance des outils de vérification consiste à optimiser ces limites, à partir de zéro ou en affinant une première série de limites proposées par l'utilisateur. La méthode fonctionne en récupérant toutes les contraintes implicites et explicites du modèle à vérifier, et en les exprimant comme un CSP sur un ensemble de variables représentant les limites à utiliser au moment de la vérification du modèle. Avec le CSP en place, ces limites sont alors calculées par un solveur en utilisant des techniques de propagation de contraintes d'intervalles. Cette méthode n'est pas optimale mais elle est sûre. Elle peut ne pas calculer les meilleures limites mais elle va conserver toutes les instances de modèle qui existent dans les limites d'origine, et qui sont donc une solution pour le problème. Finalement, ces nouvelles limites avec le modèle initial sont transmises à un outil de vérification borné pour effectuer la vérification.

A.4 Test de Transformations de Modèles

Après avoir décrit l'état de l'art sur la vérification des modèles statiques, et présenté diverses contributions pour son amélioration, dans cette section nous mettons l'accent sur le domaine du test de transformations de modèles.

A.4.1 Etat de L'Art

Essentiellement, la méthodologie pour tester une transformation de modèles est la même que celle suivie par les méthodes traditionnelles de test de programmes, c'est-à-dire:

- Déterminer les données d'entrée appropriées pour tester la transformation de modèles.
- Exécuter la transformation de modèles avec les données d'entrée obtenues à partir de la première étape.
- Analyser les sorties produites par l'exécution de la transformation de modèles pour détecter la présence éventuelle d'erreurs.

Dans cette thèse, l'attention est portée sur la première étape.

Au moment de générer des données d'entrée pour tester une transformation de modèles, et comme c'est le cas pour les méthodes traditionnelles de test de programmes, les approches les plus populaires peuvent être regroupées en deux grandes familles: approches de types "boîte noire" et "boîte blanche". Dans le cas du test "boîte noire" les données d'entrée sont générées au moyen de l'analyse de la spécification de la transformation de modèles, tandis que dans le cas du test "boîte blanche" elles sont générées au moyen de l'analyse de la structure interne de la transformation de modèles. En général les données d'entrée seront des modèles et, par conséquent, nous les appellerons "modèles de test" à partir de maintenant.

Le nombre d'approches de test boîte noire existantes est plutôt limité. Elles peuvent être divisées en deux sous-groupes en fonction de la stratégie suivie: utilisant une analyse de partition ou une sorte de langage logique.

L'analyse de partition [127] est utile pour identifier les régions non vides et disjointes de l'espace d'instance du métamodèle d'entrée de la transformation de modèles, lorsque les modèles partagent les mêmes caractéristiques. Avec cette information, il est alors possible de créer un modèle de test à partir de chaque région identifiée. La première tentative d'utilisation de l'analyse de partition pour obtenir des modèles de test sur les diagrammes de classes UML a été réalisée par Andrews et al. [128]. Cette approche a ensuite été utilisée par Fleurey et al. [129] pour proposer une approche de test boîte noire pour des transformations de modèles. Après cela, d'autres approches basées sur [129] ont été proposées [132–134]. D'autre part, les approches de test boîte noire qui représentent le problème de génération des modèles de test au moyen d'un langage logique [135, 138, 139] profitent de la présence d'un solveur pour obtenir des modèles de test.

Par rapport au nombre d'approches de test boîte noire, le nombre d'approches de test boîte blanche est encore plus réduit. Fleurey et al. proposent aussi dans [129] une approche boîte blanche pour améliorer l'approche boîte noire présentée dans le même travail. Egalement basé sur les concepts décrits dans [129], Wang et al. [150]

proposent un outil pour la génération automatique de modèles de test suivant une approche boîte blanche sur les règles de transformations de modèles écrites dans la langue de transformation Tefkat (http://tefkat.sourceforge.net/). Finalement, Küster et al. [149] proposent également trois différentes techniques de test boîte blanche pour l'analyse des transformations de modèles décrites au moyen de l'outil IBM WebSphere Business Modeler.

A.4.2 Défis et Domaines D'Amélioration

Au moment de tester une transformation de modèles, le défi le plus important est lié à la nature des données impliquées dans le processus. Les modèles ont tendance à être des structures grandes et complexes qui sont conformes à des métamodèles étant aussi grands et complexes, éventuellement complétés par des règles de validité écrites dans un langage de contraintes comme OCL. Cette complexité affecte les mécanismes de génération de modèles de test et de construction d'oracle. Dans le cas particulier de la génération de modèles de test, cette complexité est transformée en un problème de satisfaction de contraintes. En effet, cela implique normalement la recherche d'une structure de graphe qui satisfait un possiblement grand nombre de contraintes dans un, possiblement aussi grand, espace de recherche. En outre, les approches boîte noire basées sur la technique d'analyse de partition proposée par Andrews et al. [128] ne considèrent les contraintes OCL que superficiellement au moment de la construction des partitions. L'utilisation d'un langage logique a aussi des inconvénients, parce qu'il transforme la génération de modèles de test en un problème du même genre que celui étudié dans la Section A.3. Par conséquent, les défis discutés à ce moment là sont appliquées ici également. Finalement, le principal défi sur les approches boîte blanche est lié à l'absence d'homogénéité entre les langages de transformation de modèles existants.

A.4.3 Une Approche de Test Boîte Noire Basée sur L'Analyse de Partition et des Contraintes

Comme il a été mentionné précédemment, une des faiblesses des approches de test boîte noire basées sur la technique d'analyse de partition proposé par Andrews et al. [128] est qu'elles ne considèrent les contraintes OCL que superficiellement au moment de la construction des partitions. Cela limite la représentativité des modèles de test et le degré de couverture obtenu. Pour essayer d'améliorer cela, nous présentons une approche de test boîte noire dans laquelle la génération des modèles de test est gérée par l'analyse de chaque contrainte OCL du métamodèle d'entrée de la transformation de modèles. En particulier, la méthode prend ces contraintes OCL et, pour chacune d'elles, produit une nouvelle série d'expressions OCL car-

actérisant chacune une région de l'espace d'instance du métamodèle d'entrée de la transformation de modèles. En d'autres termes, toutes les instances de modèle dans une certaine région satisfont l'expression OCL générée qui caractérise cette région. Une fois les nouvelles expressions OCL générées, la méthode offre aux utilisateurs trois techniques de génération des modèles de test ayant différents niveaux d'exhaustivité. Avec le premier mode, appelé "mode simple", chacune de ces expressions OCL est passée séparément à EMFtoCSP avec le métamodèle d'entrée de la transformation pour générer un modèle de test qui caractérise chacune des régions. Avec la seconde approche, appelée "mode de partitions multiples", toutes les expressions OCL obtenues à partir de l'analyse d'une contrainte OCL dans le métamodèle d'entrée de la transformation sont combinées pour générer un nouvel ensemble d'expressions OCL représentant un ensemble de régions disjointes. A ce moment là, EMFtoCSP est utilisé de nouveau pour générer les modèles de test correspondants. Finalement, avec la méthode appelée "mode de partition unique", le même processus se répète. Mais, au lieu de combiner les expressions OCL résultant de l'analyse d'un contrainte OCL, il combine toutes les expressions OCL résultant de l'analyse de toutes les contraintes OCL dans le métamodèle d'entrée de la transformation de modèles.

A.4.4 ATLTest: Une Approche du Test Boîte Blanche pour Transformations de Modèles ATL

Également mentionné lors de l'exposition de l'état de l'art, le nombre d'approches du test boîte blanche est plutôt petit. Pour augmenter le nombre de méthodes disponibles, nous proposons une approche boîte blanche pour le test de transformations de modèles écrites en ATL. ATL est un langage de transformation de modèles populaire qui utilise OCL abondamment pour décrire comment les éléments du modèle sont transformés.

Notre proposition consiste à analyser les expressions OCL présentes dans le code source de la transformation de modèles pour construire un graphe orienté. Ce graphe contiendra toutes les informations nécessaires pour procéder à une deuxième étape dans laquelle les modèles de test sont générées. La méthode tente de reproduire la procédure suivie par les approches traditionnelles de test boîte blanche de programmes, basées sur le parcours d'un graphe de flot de contrôle pour générer des données d'entrée de tests. Cependant, comme ATL est principalement un langage déclaratif, les nœuds du graphe seront caractérisés par une expression OCL. De plus, les arcs indiqueront une dépendance de telle sorte que le nœud de destination ne peut être visité si l'expression OCL dans le nœud source n'est pas satisfaite.

Une fois le graphe généré, le parcours consiste à assigner des vraies valeurs aux expressions OCL dans les nœuds. Comme les arcs représentent une dépendance en-

tre les expressions OCL dans les nœuds connectés, l'attribution d'une valeur "faux" à une expression OCL représente une condition potentielle pour arrêter le processus. L'idée est que, chaque fois que le graphe est parcouru, un certain nombre d'expressions OCL mises à "vrai" ou "faux" seront produites. Ces expressions OCL, avec le métamodèle d'entrée de la transformation ATL, peuvent ensuite être passées à EMFtoCSP pour générer un modèle de test qui caractérise le chemin parcouru. Le processus peut être répété un nombre fixe de fois, ou bien jusqu'à ce que tous les parcours dans le graphe soient épuisés.

A.5 Conclusions

Cette thèse analyse le domaine des approches pour la vérification de modèles statiques avec le double objectif de 1) présenter des contributions pour améliorer leur efficacité et 2) de découvrir des nouveaux mécanismes qui utilisent ces approches pour tester des transformations de modèles.

Pour réaliser le premier objectif, un outil pour la vérification de modèles statiques dans la plateforme Eclipse, ainsi que deux approches pour améliorer la performance des techniques de vérification, ont été introduits. Finalement, pour réaliser le deuxième objectif, deux techniques de génération de modèles pour tester des transformations de modèles ont été proposées.

Bibliography

[1] Jones, C.B., O'Hearn, P.W., Woodcock, J.: Verified software: A grand challenge. IEEE Computer **39**(4) (April 2006) 93–95 16

[2] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional (2001) 16

[3] Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool (2012) 16, 21

[4] Dobing, B., Parsons, J.: How UML is used. Communications of the ACM **49**(5) (May 2006) 109–113 17, 35

[5] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–646 24

[6] Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125–142 24

[7] Jouault, F., Kurtev, I.: Transforming models with atl. In: Satellite Events at the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 128–138 25, 121, 125

[8] Cabot, J., Gogolla, M.: Object constraint language (OCL): A definitive guide. In: Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012. Volume 7320 of Lecture Notes in Computer Science., Springer (2012) 58–90 25

[9] Kurbel, K.E.: The Making of Information Systems: Software Engineering and Management in a Globalized World. Springer (2008) 25

[10] Garvin, D.A.: What "does product quality" really mean? Sloan Management Review **26**(1) (Fall 1984) 25â43 25

[11] Kitchenham, B., Pfleeger, S.L.: Software quality: The elusive target. IEEE Software **13**(1) (January 1996) 12–21 25

[12] McCall, J.A., Richards, P.K., Walters, G.F.: Factors in software quality. technical report. Technical Report RADC-TR-77-369, U.S. Department of Commerce (1977) 25

[13] VV.AA.: IEEE Standard Glossary of Software Engineering Terminology. IEEE Computer Society (1990) 26, 28

[14] Boehm, B.W.: Software Engineering Economics. Prentice Hall (1981) 26, 28

[15] Rushby, J.: Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International (1995) 26

[16] Gurevich, Y.: Evolving algebras 1993: Lipari guide. Specification and validation methods (1995) 9–36 27

[17] Abrial, J.R.: The B-Book: Assigning programs to meanings. Cambridge University Press (1996) 27, 43

[18] Spivey, J.M.: An introduction to z and formal specifications. Software Engineering Journal **4**(1) (January 1989) 40â50 27

[19] Mazzeo, A., Mazzocca, N., Russo, S., Savy, C., Vittorini, V.: Formal specification of concurrent systems: A structured approach. The Computer Journal **41**(3) (1998) 145–162 27

[20] Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science Journal **80**(1) (March 1991) 1–34 27

[21] Sterling, L.S., Shapiro, E.Y.: The Art of Prolog, 2nd Edition. The MIT Press (1994) 27

[22] Barendregt, H.P.: The Lambda Calculus, its Syntax and Semantics, Vol. 103 in Studies in Logic and the Foundations of Mathematics. North-Holland (1984) 27

[23] Thompson, S.: Haskell: The Craft of Functional Programming, 3rd Edition. Addison-Wesley Professional (2011) 27

[24] Klop, J.W., de Vrijer, R., (eds.), M.B.: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003) 27

[25] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999) 27

[26] Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340 27

[27] Dutertre, B., de Moura, L.: The Yices SMT solver. Technical report, SRI International (2006) 27

[28] Coq Development Team, T.: The Coq Proof Assistant Reference Manual – Version V8.0. (April 2004) 27

[29] Paulson, L.C.: Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow). Volume 828 of Lecture Notes in Computer Science. Springer (1994) 27

[30] Bowen, J.P., Butler, R.W., Dill, D.L., Glass, R.L., Gries, D., Hall, A., Hinchey, M.G., Holloway, C.M., Jackson, D., Jones, C.B., Lutz, M.J., Parnas, D.L., Rushby, J.M., Wing, J.M., Zave, P.: An invitation to formal methods. IEEE Computer **29**(4) (1996) 16–30 27

[31] Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd Edition. John Wiley & Sons, Inc. (2012) 28, 29, 125

[32] Dijkstra, E.W.: Notes on structured programming, 2nd edition. t.-h.-report 70-wsk-03. Technical report, Technical University Eindhoven (1970) 28

[33] Naik, K., Tripathy, P.: Software Testing and Quality Assurance: Theory and Practice. John Wiley & Sons, Inc. (2008) 28, 29

[34] Perry, W.E.: Effective Methods for Software Testing, 3rd Edition. Wiley Publishing, Inc. (2006) 28

[35] Beizer, B.: Software Testing Techniques, 2nd Edition. The Coriolis Group (1990) 29, 97

[36] Burnstein, I.: Practical Software Testing: A Process-Oriented Approach. Springer (2003) 30

[37] Beck, K.: Test-Driven Development By Example. Addison Wesley (2002) 30

[38] Barták, R.: Constraint programming: In pursuit of the holy grail. In: 8th Annual Conference of Doctoral Students, WDS'99, MaftyzPress (1999) 555–564 30

[39] Rossi, F., van Beek, P., (eds.), T.W.: Handbook of Constraint Programming. Foundations of Artificial Intelligence, Volume 2. Elsevier (2006) 30

[40] Sutherland, I.E.: Sketchpad, a man-machine graphical communication system. phd thesis (1963) 31

[41] Waltz, D.: Understanding line drawings of scenes with shadows. In: The Psychology of Computer Vision, McGraw-Hill Computer Science Series (1975) 31

[42] Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003) 31

[43] Jaffar, J., Lassez, J.L.: Constraint logic programming. In: 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL87, ACM (1987) 111–119 32

[44] Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence Journal **168**(1-2) (October 2005) 70–118 36, 37, 38, 39, 40, 144, 145

[45] Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2007. (2007) 547–548 37, 38, 52, 73, 79, 99, 144, 146

[46] Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Proceedings of the Workshops of the 1st International Conference on Software Testing Verification and Validation, ICST 2008, IEEE Computer Society (2008) 73–80 37, 38

[47] Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-driven slicing of UML/OCL models. In: 25th IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, ACM (2010) 185–194 38, 39

[48] Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. Electronic Communication of the European Association of Software Science and Technology (ECEASST) **24** (2009) 38, 39

[49] Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: 3rd European Conference on Model Driven Architecture-Foundations and Applications, ECMDA-FA 2007. Volume 4530 of Lecture Notes in Computer Science., Springer (2007) 17–31 38, 39, 145

[50] Balaban, M., Maraee, A.: A UML-Based method for deciding finite satisfiability in description logics. In: 21st International Workshop on Description Logics, DL 2008. Volume 353 of CEUR Workshop Proceedings., CEUR-WS.org (2008) 38, 39

[51] Queralt, A., Rull, G., Teniente, E., Farré, C., Urpí, T.: AuRUS: Automated reasoning on UML/OCL schemas. In: 29th International Conference on Conceptual Modeling, ER 2010. Volume 6412 of Lecture Notes in Computer Science., Springer (2010) 438–444 38, 39, 146

[52] Queralt, A., Teniente, E.: Reasoning on UML class diagrams with OCL constraints. In: 25th International Conference on Conceptual Modeling, ER 2006. Volume 4215 of Lecture Notes in Computer Science., Springer (2006) 497–512 38, 40

[53] Queralt, A., Teniente, E.: Decidable reasoning in UML schemas with constraints. In: 20th International Conference on Advanced Information Systems Engineering, CAiSE 2008. Volume 5074 of Lecture Notes in Computer Science., Springer (2008) 281–295 38, 40

[54] Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Transactions on Software Engineering and Methodology **21**(2) (March 2012) 13:1–13:41 38, 40

[55] Rull, G., Farré, C., Teniente, E., Urpí, T.: Providing explanations for database schema validation. In: 19th International Conference on Database and Expert Systems Applications, DEXA 2008. Volume 5181 of Lecture Notes in Computer Science., Springer (2008) 660–667 38, 40

[56] Cadoli, M., Calvanese, D., Giacomo, G.D., Mancini, T.: Finite model reasoning on UML class diagrams via constraint programming. In: 10th Congress of the Italian Association for Artificial Intelligence and Human-Oriented Computing, AI*IA 2007. Volume 4733 of Lecture Notes in Computer Science., Springer (2007) 36–47 38, 40

[57] Cadoli, M., Calvanese, D., Mancini, T.: Finite satisfiability of UML class diagrams by Constraint Programming. In: 2004 International Workshop on Description Logics, DL 2004. Volume 104 of CEUR Workshop Proceedings., CEUR-WS.org (2004) 38, 40

[58] Cadoli, M., Calvanese, D., Giacomo, G.D.: Towards implementing finite model reasoning in description logics. In: 2004 International Workshop on Description Logics, DL 2004. Volume 104 of CEUR Workshop Proceedings., CEUR-WS.org (2004) 38, 40

[59] Artale, A., Calvanese, D., Ibáñez-García, Y.A.: Checking full satisfiability of conceptual models. In: 23rd International Workshop on Description Logics, DL 2010. Volume 573 of CEUR Workshop Proceedings., CEUR-WS.org (2010) 38, 40

[60] Artale, A., Calvanese, D., Ibáñez-García, Y.A.: Full satisfiability of UML class diagrams. In: 29th International Conference on Conceptual Modeling, ER 2010. Volume 6412 of Lecture Notes in Computer Science., Springer (2010) 317–331 38, 40

[61] Artale, A., Calvanese, D., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Reasoning over extended ER models. In: 26th International Conference on Conceptual Modeling, ER 2007. Volume 4801 of Lecture Notes in Computer Science., Springer (2007) 277–292 38, 40

[62] Artale, A., Calvanese, D., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Complexity of reasoning in entity relationship models. In: 20th International Workshop on Description Logics, DL 2007. Volume 250 of CEUR Workshop Proceedings., CEUR-WS.org (2007) 38, 40

[63] Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams is EXPTIME-hard. In: 2003 International Workshop on Descrip-

tion Logics, DL 2003. Volume 81 of CEUR Workshop Proceedings., CEUR-WS.org (2003) 38, 40

[64] Berardi, D.: Using DLs to reason on UML class diagrams. In: Workshop on Applications of Description Logics, ADL-2002. Volume 63 of CEUR Workshop Proceedings., CEUR-WS.org (2002) 38, 40

[65] Calì, A., Calvanese, D., Giacomo, G.D., Lenzerini, M.: A formal framework for reasoning on UML class diagrams. In: 13th International Symposium Foundations of Intelligent Systems, ISMIS 2002. Volume 2366 of Lecture Notes in Computer Science., Springer (2002) 503–513 38, 40

[66] Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: A decidable (yet expressive) fragment of OCL. In: 25th International Workshop on Description Logics, DL 2012. Volume 846 of CEUR Workshop Proceedings., CEUR-WS.org (2012) 38, 40, 145

[67] Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering Journal **73** (March 2012) 1–22 38, 40

[68] Formica, A.: Finite satisfiability of integrity constraints in Object-Oriented database schemas. IEEE Transactions on Knowledge and Data Engineering **14**(1) (January/February 2002) 123–139 38, 41

[69] Formica, A.: Satisfiability of Object-Oriented database constraints with set and bag attributes. Information Systems Journal **28**(3) (May 2003) 213–224 38, 41

[70] Brucker, A.D., Wolff, B.: HOL-OCL: A formal proof environment for UML/OCL. In: 11th International Conference on Fundamental Approaches to Software Engineering, FASE 2008. Volume 4961 of Lecture Notes in Computer Science., Springer (2008) 97–100 38, 41, 144

[71] Brucker, A.D., Doser, J., Wolff, B.: An MDA framework supporting OCL. Electronic Communication of the European Association of Software Science and Technology (ECEASST) **5** (2006) 38, 41

[72] Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zurich (2006) 38, 41

[73] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 436–450 38, 41, 73, 144

[74] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling **9**(1) (January 2010) 69–86 38, 41

[75] Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and System Modeling **4**(4) (November 2005) 386–398 38, 42, 144

[76] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based specification environment for validating UML and OCL. Science of Computer Programming **69**(1-3) (December 2007) 27–34 38, 42

[77] Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: 3rd International Conference on Tests and Proofs, TAP 2009. Volume 5668 of Lecture Notes in Computer Science., Springer (2009) 90–104 38, 42

[78] Gogolla, M., Hamann, L., Kuhlmann, M.: Proving and visualizing OCL invariant independence by automatically generated test cases. In: 4th International Conference on Tests and Proofs, TAP 2010. Volume 6143 of Lecture Notes in Computer Science., Springer (2010) 38–54 38, 42

[79] Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In: 49th International Conference on Objects, Models, Components and Patterns, TOOLS 2011. Volume 6705 of Lecture Notes in Computer Science., Springer (2011) 290–306 38, 42

[80] Kuhlmann, M., Gogolla, M.: Strengthening SAT-Based validation of UML/OCL models by representing collections as relations. In: 8th European Conference on Modelling Foundations and Applications, ECMFA 2012. Volume 7349 of Lecture Notes in Computer Science., Springer (2012) 32–48 38, 42

[81] Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems, MODELS 2012. Volume 7590 of Lecture Notes in Computer Science., Springer (2012) 415–431 38, 42, 73, 89

[82] Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: Design, Automation and Test in Europe, DATE 2010, IEEE (2010) 1341–1344 38, 42, 73

[83] Soeken, M., Wille, R., Drechsler, R.: Encoding OCL data types for SAT-Based verification of UML/OCL models. In: 5th International Conference on Tests and Proofs, TAP 2011. Volume 6706 of Lecture Notes in Computer Science., Springer (2011) 152–170 38, 42

[84] Rahim, L.A.: Mapping from OCL/UML metamodel to PVS metamodel. In: International Symposium on Information Technology, ITSim 2008. Volume 1., IEEE Computer Society (2008) 1–8 38, 43

[85] Beckert, B., Keller, U., Schmitt, P.H.: Translating the object constraint language into First-Order predicate logic. In: 2nd Verification Workshop, VERIFY 2002. Volume 2002/07 of DIKU Technical Reports., Department of Computer Science, University of Copenhagen, DIKU (2002) 113–123 38, 43

[86] Roe, D., Broda, K., Russo, A., Russo, R.: Mapping UML models incorporating OCL constraints into Object-Z. Technical report, Department of Computer Science, Imperial College London (2003) 38, 43

[87] Kamenoff, R.M., Lévy, N.: Using B formal specifications for analysis and verification of UML/OCL models. In: Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language. (2002) 38, 43, 144

[88] Ali, T., Nauman, M., Alam, M.: An accessible formal specification of the UML and OCL meta-model in Isabelle/HOL. In: 2007 IEEE International Multitopic Conference, INMIC 2007, IEEE Computer Society (2007) 1–6 38, 44

[89] Szlenk, M.: Formal semantics and reasoning about UML class diagram. In: 1st International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2006, IEEE Computer Society (2006) 51–59 38, 44

[90] Lenzerini, M., Nobili, P.: On the satisfiability of dependency constraints in entity-relationship schemata. Information Systems Journal **15**(4) (1990) 453–461 39

[91] Farré, C., Teniente, E., Urpí, T.: Checking query containment with the CQC method. Data & Knowledge Engineering Journal **53**(2) (May 2005) 163–223 39

[92] Farré, C., Rull, G., Teniente, E., Urpí, T.: SVTe: A tool to validate database schemas giving explanations. In: 1st International Workshop on Testing Database Systems, DBTest 2008, ACM (2008) 9 40

[93] Calvanese, D.: Finite model reasoning in description logics. In: 5th International Conference on Principles of Knowledge Representation and Reasoning, KR 96, Morgan Kaufmann (1996) 292–303 40

[94] Formica, A., Missikoff, M.: Integrity constraints representation in Object-Oriented databases. In: 1st International Conference on Information and Knowledge Management: Expanding the Definition of “Database”, CIKM

92, Selected Papers. Volume 752 of Lecture Notes in Computer Science., Springer (1993) 69–85 41

[95] Beeri, C.: A formal approach to Object-Oriented databases. Data & Knowledge Engineering **5** (October 1990) 353–382 41

[96] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science. Springer (2002) 41, 44

[97] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006) 41, 74, 98

[98] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007. Volume 4424 of Lecture Notes in Computer Science., Springer (2007) 632–647 42, 89

[99] Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. SRI International, Computer Science Laboratory. (1999) 43

[100] Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Eclipse development tools for Epsilon. In: Eclipse Summit Europe, Eclipse Modeling Symposium. (2006) 43

[101] Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of Lecture Notes in Computer Science. Springer (2006) 43

[102] Duke, R., Rose, G., Smith, G.: Object-Z: A specification language advocated for the description of standards. Computer Standards And Interfaces **17**(5-6) (September 1995) 511–533 43

[103] Smith, G.: The Object-Z Specification Language (Advances in Formal Methods). Springer (1999) 43

[104] Spivey, J.M.: The Z notation: A reference manual. Prentice-Hall (1989) 43

[105] Kim, S.K., Carrington, D.: A formal mapping between UML models and Object-Z specifications. In: ZB 2000: Formal Specification and Development in Z and B. 1st International Conference of B and Z Users. Volume 1878 of Lecture Notes in Computer Science., Springer (2000) 2–21 43

[106] Wahler, M., Basin, D.A., Brucker, A.D., Koehler, J.: Efficient analysis of pattern-based constraint specifications. Software and System Modeling **9**(2) (April 2010) 225–255 47

[107] Büttner, F., Cabot, J.: Lightweight string reasoning for OCL. In: 8th European Conference on Modelling Foundations and Applications, ECMFA

2012. Volume 7349 of Lecture Notes in Computer Science., Springer (2012) 244–258 52, 147

[108] Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using Constraint Programming. Journal of Systems and Software (2014 [Published online]) 53, 79

[109] Gogolla, M.: Tales of ER and RE syntax and semantics. In: Transformation Techniques in Software Engineering, 17.-22. Volume 05161 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (April 2005) 53

[110] Régin, J.C.: Generalized arc consistency for global cardinality constraint. In: Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Volume 1, AAAI Press / The MIT Press (1996) 209–215 60

[111] Gent, I.P., Smith, B.: Symmetry breaking during search in constraint programming. In: 14th European Conference on Artificial Intelligence, ECAI 2000, IOS Press (2000) 599–603 61

[112] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 168–176 74

[113] Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-soft: Software verification platform. In: Computer Aided Verification, 17th International Conference, CAV 2005. Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 301–306 74

[114] Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. ACM Transactions on Programming Languages and Systems **29**(3) (2007) 74

[115] Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with SAT. In: ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, ACM (2006) 109–120 74

[116] Dolby, J., Vaziri, M., Tip, F.: Finding bugs efficiently with a SAT solver. In: 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2007, ACM (2007) 195–204 74

[117] Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: Taco: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. IEEE Transactions on Software Engineering **39**(9) (2013) 1283–1307 74

[118] Nijjar, J., Bultan, T.: Bounded verification of ruby on rails data models. In: 20th International Symposium on Software Testing and Analysis, ISSTA 2011, ACM (2011) 67–77 74

[119] Yu, F., Bultan, T., Peterson, E.: Automated size analysis for OCL. In: 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2007, ACM (2007) 331–340 75, 79, 80

[120] Apt, K.R., Wallace, M.: Constraint Logic Programming using ECLiPSe. Cambridge University Press (2007) 77, 89, 141

[121] Demuth, B.: OCL by example (2009) MINE Summer School (Course Material), http://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf. 89

[122] Zimányi, E.: Object Constraint Language (2009) Object-Oriented Analysis and Design (Course Material), http://cs.ulb.ac.be/public/_media/teaching/infoh302/oclnotes.pdf. 89

[123] Ab.Â Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software and System Modeling (June 2013 [Published online]) 1–26 96

[124] Selim, G.M.K., Cordy, J.R., Dingel, J.: Model transformation testing: The state of the art. In: 1st Workshop on the Analysis of Model Transformations, ACM 2012, ACM (2012) 21–26 96

[125] Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM **53**(6) (June 2010) 139–143 96

[126] Weyuker, E.J.: The evaluation of program-based software test data adequacy criteria. Communications of the ACM **31**(6) (June 1988) 668–675 97

[127] Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating functional tests. Communications of the ACM **31**(6) (June 1988) 676–686 97, 107, 149

[128] Andrews, A.A., France, R.B., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability **13**(2) (April/June 2003) 95–127 98, 103, 149, 150

[129] Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: 1st International Workshop on Model, Design and Validation, MODEVA 2004. (2004) 29–40 98, 100, 120, 149

[130] Baudry, B., Fleurey, F., Jézéquel, J.M., Traon, Y.L.: Automatic test cases optimization using a bacteriological adaptation model: Application to .NET

component. In: 17th International Conference on Automated Software Engineering, ASE 2002, IEEE (2002) 253–256 98

[131] Baudry, B., Fleurey, F., Jézéquel, J.M., Traon, Y.L.: Genes and bacteria for automatic test cases optimization in the .NET environment. In: 13th International Symposium on Software Reliability Engineering, ISSRE 2002), IEEE (2002) 195–206 98

[132] Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: 17th International Symposium on Software Reliability Engineering, ISSRE 2006), IEEE (2006) 85–94 98, 149

[133] Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. Software and System Modeling **8**(2) (April 2009) 185–203 98, 149

[134] Lamari, M.: Towards an automated test generation for the verification of model transformations. In: ACM Symposium on Applied Computing, SAC 2007, ACM (2007) 998–1005 98, 101, 149

[135] Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select models for model transformation testing. In: 1st International Conference on Software Testing, Verification and Validation, ICST 2008, IEEE (2008) 328–337 98, 149

[136] Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: 2nd International Conference on Theory and Practice of Model Transformations, ICMT 2009. Volume 5563 of Lecture Notes in Computer Science. (2009) 148–164 99

[137] Mottu, J.M., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: 2nd European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA 2006. Volume 4066 of Lecture Notes in Computer Science., Springer (2006) 376–390 99

[138] Fiorentini, C., Momigliano, A., Ornaghi, M., Poernomo, I.: A constructive approach to testing model transformations. In: 3rd International Conference on Theory and Practice of Model Transformations, ICMT 2010. Volume 6142 of Lecture Notes in Computer Science., Springer (2010) 77–92 99, 101, 149

[139] Guerra, E.: Specification-driven test generation for model transformations. In: 5th International Conference on Theory and Practice of Model Transformations, ICMT 2012. Volume 7307 of Lecture Notes in Computer Science., Springer (2012) 40–55 99, 102, 120, 149

[140] Guerra, E., Soeken, M.: Specification-driven model transformation testing. Software & Systems Modeling (August 2013) 1–22 99, 102

[141] Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: A visual specification language for model-to-model transformations. In: Proceddings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010, IEEE (2010) 119–126 99, 102

[142] Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: Engineering model transformations with transML. Software & Systems Modeling **12**(3) (2013) 555–577 99

[143] Gogolla, M., Vallecillo, A.: *Tract*able model transformation testing. In: 7th European Conference on Modelling Foundations and Applications, ECMFA 2011. Volume 6698 of Lecture Notes in Computer Science., Springer (2011) 221–235 100, 101

[144] Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012. Volume 7320 of Lecture Notes in Computer Science., Springer (2012) 399–437 100, 101

[145] Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Traon, Y.L.: Model transformation testing challenges. In: ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing, IMDT 2006. (2006) 100

[146] Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: OCL and Model Driven Engineering Workshop. (2004) 100

[147] Wimmer, M., Burgueño, L.: Testing M2T/T2M transformations. In: Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013. Volume 8107 of Lecture Notes in Computer Science., Springer (2013) 203–219 100

[148] Burgueño, L., Wimmer, M., Troya, J., Vallecillo, A.: Tractstool: Testing model transformations based on contracts. In: Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013). Volume 1115 of CEUR Workshop Proceedings., CEUR-WS.org (2013) 76–80 100

[149] Küster, J.M., Abd-El-Razik, M.: Validation of model transformations - first experiences using a white box approach. In: 3rd Workshop on Model Design and Validation, MODEVA 2006. (2006) 193–204 100, 150

[150] Wang, J., Kim, S.K., Carrington, D.: Automatic generation of test models for model transformations. In: 19th Australian Conference on Software Engineering, ASWEC 2008, IEEE (2008) 432–440 100, 149

[151] Quillan, J.A.M., Power, J.F.: White-box coverage criteria for model transformations. In: 1st International Workshop on Model Transformation with ATL. (2009) 101

[152] Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. Electronic Communications of the EASST **24** (2009) 102

[153] Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proceedings of the OCL and Model Driven Engineering Workshop. (2004) 102

[154] Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: oracle issue. In: Workshops Proceedings of the 1st International Conference on Software Testing Verification and Validation, ICST 2008, IEEE Computer Society (2008) 105–112 102

[155] Mottu, J.M., Baudry, B., Traon, Y.L.: Reusable mda components: A testing-for-trust approach. In: 9th International Conference on Model Driven Engineering Languages and Systems, MODELS 2006. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 589–603 102

[156] Kolovos, D.S., Paige, R.F., Polack, F.A.: Model comparison: A foundation for model composition and model transformation testing. In: Proceedings of the 2006 International Workshop on Global Integrated Model Management. GaMMa 06, ACM (2006) 13–20 102

[157] Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: Model-Driven Software Development - Research and Practice in Software Engineering, Springer (2005) 219–236 102

[158] Finot, O., Mottu, J.M., Sunyé, G., Attiogbé, C.: Partial test oracle in model transformation testing. In: 6th International Conference on Theory and Practice of Model Transformations, ICMT 2013. Volume 7909 of Lecture Notes in Computer Science., Springer (2013) 189–204 102

[159] García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I.: Eunit: A unit testing framework for model management tasks. In: 14th International Conference on Model Driven Engineering Languages

and Systems, MODELS 2011. Volume 6981 of Lecture Notes in Computer Science., Springer (2011) 395–409 103

[160] Giner, P., Pelechano, V.: Test-driven development of model transformations. In: 12th International Conference on Model Driven Engineering Languages and Systems, MODELS 2009. Volume 5795 of Lecture Notes in Computer Science., Springer (2009) 748–752 103

[161] McGill, M.J., C.Cheng, B.H.: Test-driven development of a model transformation with jemtte. East (2007) 1–17 103

[162] Kobler, J., Schöning, U., Toran, J.: The Graph Isomorphism Problem: Its Structural Complexity. Birkhäuser (1993) 104

[163] Gogolla, M., Richters, M.: Expressing UML class diagrams properties with OCL. In: Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Volume 2263 of Lecture Notes in Computer Science., Springer (2002) 85–114 119

[164] McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering **2**(4) (1976) 308–320 142

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Thèse de Doctorat

Carlos A. GONZÁLEZ

Vérification Pragmatique de Modèles

Pragmatic Model Verification

Résumé
L'Ingénierie Dirigée par les Modèles (IDM) est une approche populaire pour le développement logiciel qui favorise l'utilisation de modèles au sein des processus de développement. Dans un processus de développement logiciel base sur l'IDM, le logiciel est développé en créant des modèles qui sont transformés successivement en d'autres modèles et éventuellement en code source. Quand l'IDM est utilisée pour le développement de logiciels complexes, la complexité des modèles et des transformations de modèles augmente, risquant d'affecter la fiabilité du processus de développement logiciel ainsi que le logiciel en résultant.
Traditionnellement, la fiabilité des logiciels est assurée au moyen d'approches pour la vérification de logiciels, basées sur l'utilisation de techniques pour l'analyse formelle de systèmes et d'approches pour le test de logiciels. Pour assurer la fiabilité du processus IDM de développement logiciel, ces techniques ont en quelque sorte été adaptées pour essayer de s'assurer la correction des modèles et des transformations de modèles associées. L'objectif de cette thèse est de fournir de nouveaux mécanismes améliorant les approches existantes pour la vérification de modèles statiques, et d'analyser comment ces approches peuvent s'avérer utiles lors du test des transformations de modèles.

Mots clés
Ingénierie Dirigée par les Modèles, Vérification de Modèles, Test de Transformations de Modèles.

Abstract
Model-driven Engineering (MDE) is a popular approach to the development of software which promotes the use of models as first-class citizens in the software development process. In a MDE-based software development process, software is developed by creating models to be successively transformed into another models and eventually into the software source code. When MDE is applied to the development of complex software systems, the complexity of models and model transformations increase, thus risking both, the reliability of the software development process and the soundness of the resulting software. Traditionally, ensuring software correctness and absence of errors has been addressed by means of software verification approaches, based on the utilization of formal analysis techniques, and software testing approaches. In order to ensure the reliability of MDE-based software development processes, these techniques have somehow been adapted to try to ensure correctness of models and model transformations. The objective of this thesis is to provide new mechanisms to improve the landscape of approaches devoted to the verification of static models, and analyze how these static model verification approaches can be of assistance at the time of testing model transformations.

Key Words
Model Driven Engineering, Static Model Verification, Model Transformation Testing.