Automated Metamodel Instance Generation Satisfying Quantitative Constraints

Wu Hao(吴昊)

Supervisors: Dr. Rosemary Monahan and Dr. James F. Power

Department of Computer Science

National University of Ireland, Maynooth

A thesis submitted for the degree of

Doctor of Philosophy

October, 2013

I would like to dedicate this thesis to every one who helped me during my PhD studies.

Acknowledgements

I consider this section as one of the most important parts of this thesis because without any one of you mentioned below this thesis would never have been possible.

First and foremost, I would like to thank my supervisors Dr. Rosemary Monahan and Dr. James F. Power for their guidance during the last four years. Their critical and strict attitude towards my work are the key factors for me to keep up till the very end of my PhD studies. So, a huge thank you to both of you.

I would like to exponentially extend my gratitude to those of you who helped me during my sickness in 2011. They include: Jimmy Gorman, Joe Timoney, Lu Bing Bing(卢斌斌), Deirdre Dunne, Ying Fang Li(应方立), Yang Wen Bai(杨文白), Brendan Ashe, Zhang Xin(张欣), Chen Yu Ying(陈昱颖), Chen Shuang(陈霜), Stephen Brown and many of who visited me when I was in hospital. It was the darkest time in my PhD studies, and probably also in my life so far. Fortunately, with the encouragement and support from all of you, I was able to quickly step out the darkness and start to recover. Therefore, all of you really deserve my biggest thanks.

I also would like to thank those who shared an apartment with me during my last year studies. They include: Chen Xiao Ming(陈晓敏), Emma Jackson, Michael Roche, Holly Henning and Jamie Mac Phillips. Thanks for all the nice conversations during the dinner time in the living room, which really helped me get relaxed after my daily stressful work.

Finally, I would like to give my deepest gratitude to my loving parents (袁美萍, 吴国弟). Your greatest support is the ultra power supply for finishing this thesis, and it is far beyond any physical distance and time difference.

Abstract

Metamodels are the core of the metamodeling approach and widely used in model driven architecture. The high abstraction level provided by metamodels makes the metamodeling approach popular among software modelers and language designers. However, the metamodeling approach has one big drawback: it does not support instance generation. Instances are particularly important for software modelers and language designers to test or verify their metamodels. Unfortunately, automatically generating metamodel instances is a very challenging task. Furthermore, the generated instances should ideally cover a more generic or specific feature of a metamodel as required by modelers for different purposes.

This thesis presents a solution that combines both graph representation and Satisfiability Modulo Theories (SMT) to the problem of metamodel instance generation. The solution consists of two approaches, the first approach presents a new foundation for generating metamodel instances by translating a metamodel to an *SMT* problem via a *bounded graph* representation. The second approach investigates generating meaningful metamodel instances by using two new techniques. The first technique generates instances that meet *partition-based coverage criteria* by using criteria formulas to further constrain the entire generation process. The second technique generates instances that satisfy *graph-property based criteria* by introducing different scenarios. These two approaches have been prototyped into a new tool, named ASMIG, to demonstrate the feasibility of automatic metamodel instance generation.

Simplicity is prerequisite for reliability. Edsger.W.Dijkstra, 1975

Contents

List of Figures

Chapter 1

Introduction

The motivation of this research is to assist software engineers using metamodels for software design in Model Driven Engineering. These metamodels capture important aspects of the software that allow software engineers to refine their design at higher levels of abstraction. However, a central problem of using metamodels for modelling software is that it is difficult to automatically generate metamodel instances. This is because each valid metamodel instance must satisfy many constraints. The research presented in this thesis argues that an advantageous automated approach for producing metamodel instances can be derived from the combination of graph representation with Satisfiability Modulo Theories (SMT). In particular, the approach can be tailored to generate meaningful metamodel instances in a relatively straightforward manner.

1.1 Model Driven Engineering

The idea of working with Model Driven Engineering (MDE) is to use models as primary artifacts when software engineers are developing software. Before writing any lines of code, a model that represents an aspect of a system is built and the actual program may be automatically generated from the model. This provides software engineers with a complementary approach to generate code, being abstract and automated over the traditional manual code writing. Much research has been focused on devising methods for integrating MDE into the software engineering tool-kit [Alonso et al., 2007; Heidenreich et al., 2010b; Philippi, 2006]. The interaction between different development tasks now varies from writing different pieces of code to depicting different types of models [Jäger et al., 1999; Larman, 2003]. MDE not only changes the process of software development but also raises the software design to a higher abstraction level [Atkinson and Kühne, 2003; Baker et al., 2005; Elaasar and Neal, 2013].

1.2 Model and Metamodel

In MDE, models are key structures for building a complex system, where software engineers use them to represent different aspects of a system at an abstract level. Different types of models capture different aspects of a system. For example, models can be used to describe a system's structural or dynamic behaviours. To a software engineer, interacting and refining these models relieves the complexity of a system by removing a large amount of code which may be automatically generated from these models. A program automatically generated from a model creates real objects during its execution, and these objects created are *instances* of the model. The introduction of metamodeling, a higher abstraction technique to MDE, allowed software engineers to describe and refine their designs at an even higher level [Object Management Group, 2011a]. To a software engineer, a *model* is an abstract representation of an aspect of a system, whereas a *metamodel* describes a higher abstraction: a metamodel is a model that describes the syntax for a set of models.

Since a metamodel captures a set of models that have the same syntax, an *instance* of a metamodel is thus a model. To illustrate this concept, Figure 1.1 shows an example of a model and metamodel. In Figure 1.1 at level 1, the model $Person$ conforms to a metamodel which is defined at level 2. This higher level metamodel has two metaclasses, *Class* and *Attribute* (the field *age* conforms to *Attribute*). The conformance here means that the model defined in the lower level is an *instance* of the model defined in the level above. Thus, the $Person$ model defined at level 1 is an instance of its metamodel defined at level 2. For example, $Person$ is an instance of $Class$, and age is an instance of $Attribute$.

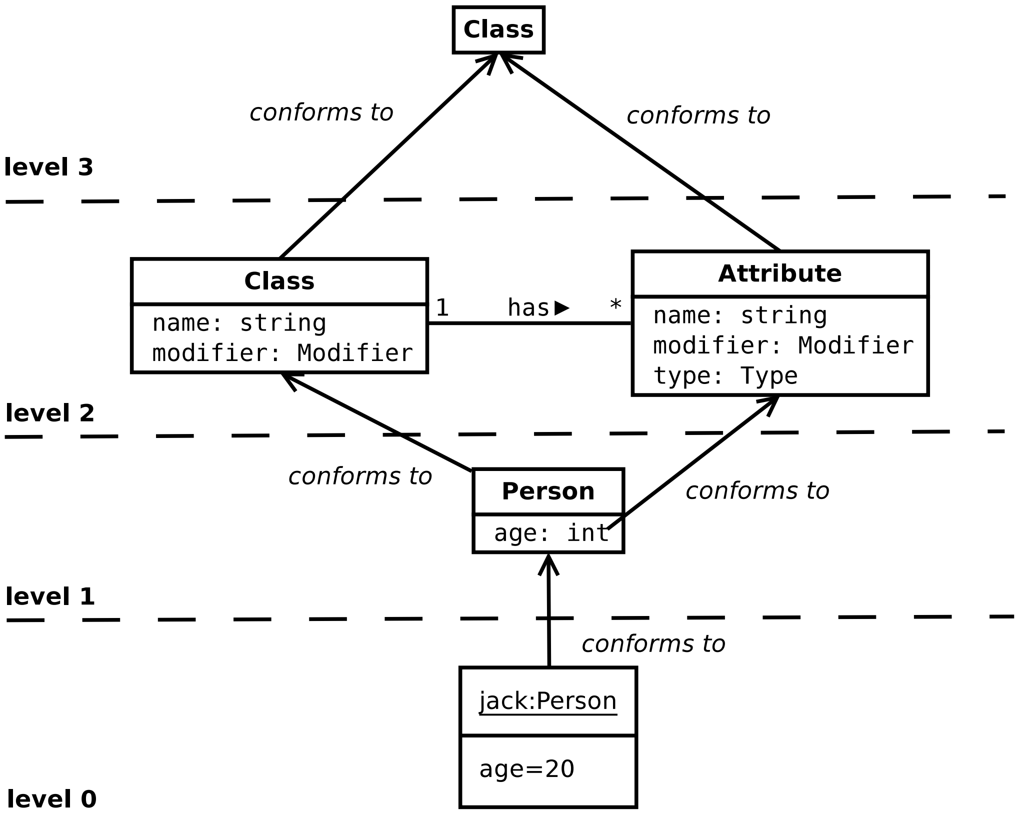


Figure 1.1: An example of model and its metamodel. This is the four-level MOF architecture, and from the top down it describes a meta-metamodel (level 3), metamodel (level 2), model (level 1) and an object (level 0).

1.3 MetaObject Facility

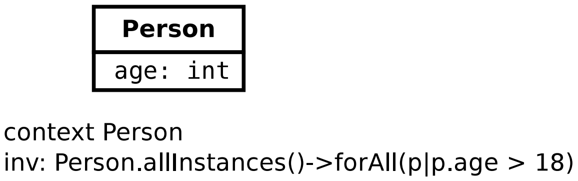
The MetaObject Facility (MOF) is a standard formalism for constructing metamodels in Model Driven Architecture (MDA), and it consists of four levels for specifying models [Object Management Group, 2011a]. Each level represents a different abstraction level, where an upper level specifies a higher abstraction than a lower level does. In Figure 1.1, at the top level (level 3) is the meta-metamodel used by MOF for building a metamodel (level 2). For example, in Figure 1.1 the meta-metamodel *Class* at level 3 defines the metamodel *Class* and *Attribute* at level 2. In other words, metamodel *Class* and *Attribute* are instances of meta-metamodel *Class*. Similarly, the model at level 1 is described by its metamodel at level 2. Models at level 0 are called objects, and they represent objects in the real-world. For example, the model *jack* with an age of 20 in Figure 1.1 represents an object that is an instance of the *Person* model.

1.4 Metamodels and UML Class Diagrams

The Unified Modeling Language (UML) is a visual modeling language that is widely used by software engineers to depict their system [Object Management Group, 2011c,d]. The UML diagrams are grouped into two main types: structural diagrams and behavioural diagrams. Since a metamodel describes a structural view of the models, this can also be captured by one type of UML structural diagram: the UML class diagram. Therefore, a metamodel can be depicted as a UML class diagram. For example, in Figure 1.1, the metamodel (level 2) for *Person* is depicted as a UML class diagram, which has 2 classes, a list of attributes in each class and a binary association.

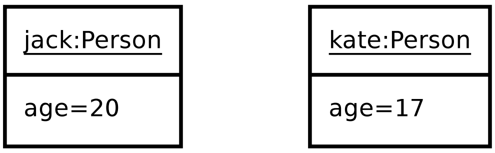
1.5 Object Constraint Language

The Object Constraint Language (OCL) is a declarative language that is used to express constraints or queries over a metamodel [Object Management Group, 2012b]. OCL uses a *first order logic* (FOL) like syntax to quantify a set of objects such as the *forAll* operator in Figure 1.2. However, OCL has more



```
context Person
inv: Person.allInstances()->forAll(p|p.age > 18)
```

Figure 1.2: A metamodel with an OCL invariant indicating a person's age must be over 18.



(a) (b)

Figure 1.3: Two instances: Figure 1.3(a) is a valid instance for metamodel in Figure 1.2, Figure 1.3(b) is an invalid instance.

expressiveness than FOL for specifying advanced properties about a metamodel, such as using a closure operator to return results from the elements of the elements of a collection data type. The main purpose of using OCL is to specify invariants and pre/post conditions for the state of a system. An OCL invariant defined on a metamodel specifies what any instances of the metamodel have to obey. For example, Figure 1.2 shows an OCL invariant for a *Person* metamodel: this invariant simply requires that every person's age (instance) must be over 18, such as in Figure 1.3(a). Any person with an age that does not satisfy this invariant is simply not a valid instance of the metamodel. For example, Figure 1.3(b) shows an invalid instance.

1.6 Problem Statement

A central question with the metamodeling approach is: given a metamodel and a set of constraints defined in OCL, how can one (automatically) generate valid instances? Here valid instances mean that the generated instances have to conform to the metamodel itself and any defined OCL constraints. Without automatic instance generation, modelers have to manually design instances (test cases or a test suite) to test or verify the correctness of their metamodels, and such manually designed instances are infeasible when the structure of a metamodel becomes very complex and when a large number of OCL constraints are defined. Furthermore, generating metamodel instances satisfying some properties like coverage criteria is very useful for modelers to test specific metamodel features such as inheritance relationships or associations.

1.7 Motivation

Being able to automatically generate such a test suite would have two main benefits. First, examining the automatically-generated test cases would help to develop the modeler's understanding of the metamodel, and help to increase confidence in its validity. Second, since the metamodel can be used to specify a programming language, the generated test cases should be valid programs from that language, and these can then be used as test inputs for tools that process the language [Wu et al., 2010].

1.8 Challenges

In general, generating metamodel instances is undecidable if no restrictions on the bounds are given [Balaban and Maraee, 2013; Wu et al., 2013]. This is because a metamodel can have OCL as its constraints, OCL is more expressive than FOL and FOL itself is undecidable [Ben-Ari, 2012; Boolos et al., 2003; Cabot et al., 2008]. The challenge here is not only that the instances have to satisfy the metamodel structural constraints but that they also need to satisfy the constraints specified in OCL over a metamodel. Furthermore, ideally each generated metamodel instance should also meet some properties such as the coverage criteria required by its users. A test suite can thus be formed by enumerating these instances.

1.9 Summary of Contributions

This thesis focuses on these challenges, and contributes a new solution that consists of two approaches for generating metamodel instances automatically. A systematic literature review (Chapter 2) on existing techniques used for generating metamodel instances and background knowledge (Chapter 3) used through our two approaches is first presented. Chapters 4, 5 and 6 present our solution, consisting of two approaches to solving the problem and form the main contributions of this thesis [Wu et al., 2012]. These approaches are:

1. A novel approach that constructs a new foundation that naturally supports metamodeling and translates the metamodel instance generation problem to an SMT problem. This translation scheme from metamodel to SMT uses a new graph concept; a bounded attributed type graph with inheritance, and this concept acts as an intermediate representation during the translation. The translation also provides a set of rules for translating a subset of OCL constraints to SMT (Chapter 4) [Wu et al., 2013].
2. An approach that uses two new techniques to extend the new foundation by generating meaningful metamodel instances in two different ways: satisfying partition-based coverage criteria (Chapter 5) and graph-properties based criteria (Chapter 6).

These approaches have been prototyped in a tool: *A Small Metamodel Instance Generator (ASMIG).* ASMIG is a new tool that is designed to naturally support metamodels, fully automate the metamodel instance generation process and utilise an advanced decision procedure (Z3) [De Moura and Bjørner, 2008] as its back-end reasoning engine. ASMIG is a relatively small tool that consists of about $23,000$ lines of code (LOC) and with around $12,000$ LOC dedicated to its key components. Figure 1.4 shows the main components of ASMIG and the process of generating metamodel instances. We have evaluated ASMIG against a considerable number of metamodels of different sizes, and the evaluation from ASMIG reveals both strengths and limitations of this research. This leads to the discussion of future research in this area (Chapter 7).

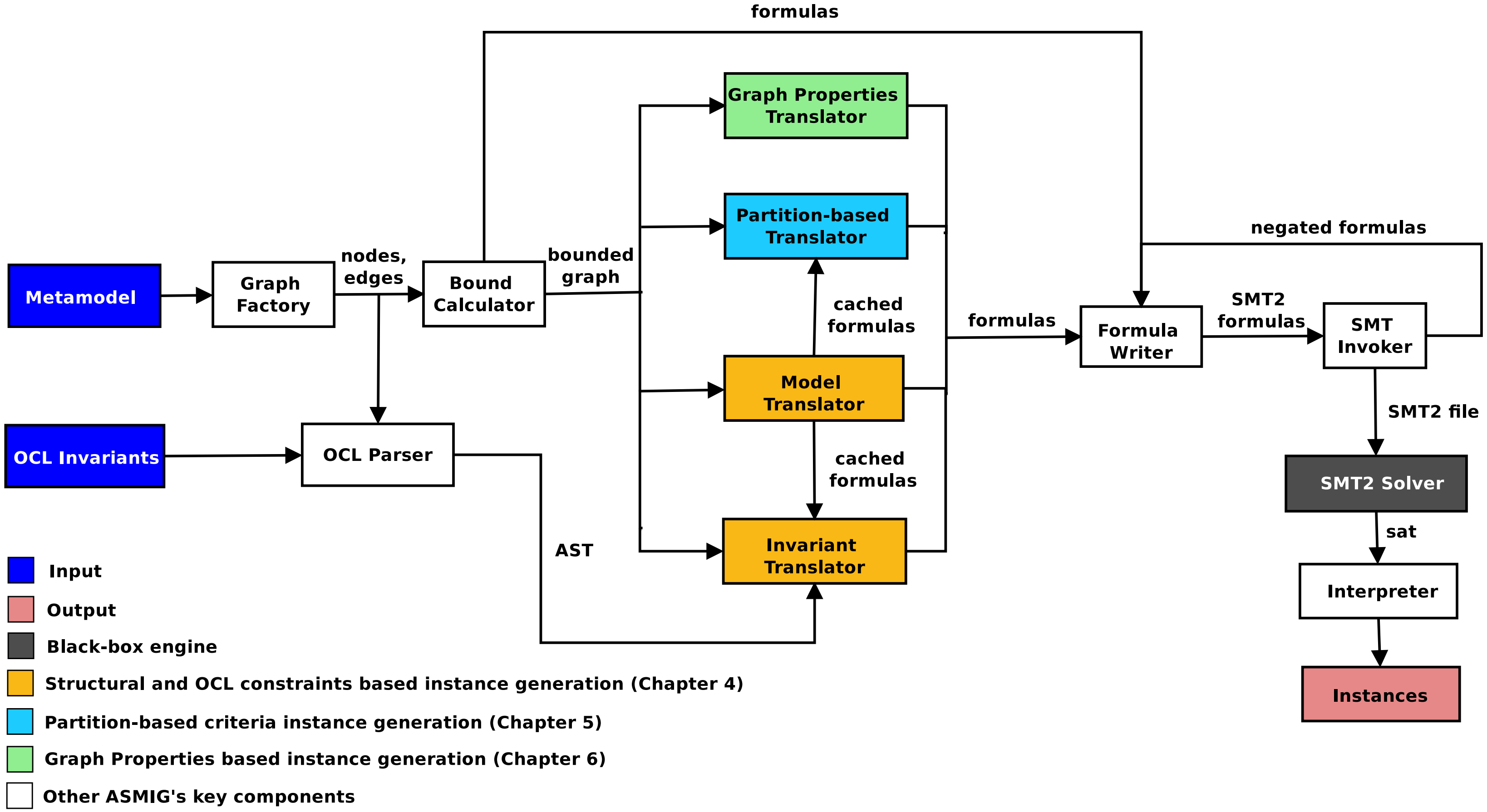


Figure 1.4: The approach and techniques presented in Chapter 4, 5 and 6 have been implemented as a new tool: A Small Metamodel Instance Generator (ASMIG). Not only can ASMIG automatically generate metamodel instances satisfying structural and OCL constraints (described in Chapter 4) but also with consideration for partition and graph based criteria (described in Chapters 5 and 6). The generated formulas from ASMIG conform to the SMT-Lib Version 2.0 (SMT2) standard, and input to the SMT2 solver (that acts as a black-box engine) for solving the formulas, and each successful assignment for formulas is converted back into an instance of a metamodel.

Chapter 2

A Systematic Literature Review of Metamodel Instance Generation Techniques

Although generating metamodel instances can be surprisingly difficult, much research on generating instances using different approaches has been proposed. However, it is still difficult for modelers and researchers to choose the most suitable technique among them to meet their requirements because a lack of reporting and comparisons for identifying the advantages and drawbacks of each technique. In this chapter, we present a systematic literature review that discusses these approaches, outlines the advantages and drawbacks of each approach, and identifies the deficiencies in the current techniques and outline how they could be rectified.

2.1 Review Research Methods

We first present the steps taken for our systematic literature review. The steps are in accordance with the systematic review guidelines given in the paper by Kitchenham [2004].

1. Identify the need for a systematic literature review.

2. Formulate the review research questions.

3. Conduct a comprehensive, exhaustive search for the primary studies.
4. Assess the quality of the included studies.
5. Extract the data from each included study.
6. Summarise and synthesise the results of the study.
7. Interpret the results and determine their applicability.
8. Write-up the study as a report.

To our best knowledge, no literature review has ever been conducted to identify the advantages and disadvantages of each approach to metamodel instance generation. The aim of this literature review is to analyse the state of the art in the field of metamodel instance generation, and identify any existing research gaps.

2.2 Defining The Research Questions

In this section, we show that we derive our four research questions from three angles: the key research domains that have techniques for generating metamodel instances, the main theoretical frameworks, and the tools available for generating metamodel instances.

To identify any key research domains that have techniques for generating metamodel instances without any prior knowledge is difficult. Therefore, our starting point is to perform an initial study on the topic. For this reason, we chose Purdom's sentence generation algorithm as this study [Purdom, 1972]. This algorithm generates sample sentences from a given grammar. We chose this paper because it uses a very similar idea to metamodeling, but instead it uses a grammar. This paper possibly can be considered one of the oldest techniques dealing with instance generation (sentence generation) in the grammar domain. More importantly, what we learn from this initial study is that it reveals that metamodel instance generation may come from a grammar research domain. This is because a metamodel can be used for capturing an abstract syntax tree of a programming language. Therefore, by considering the abstraction level provided by a metamodel, we realise that the research domains that have techniques for generating metamodel instances may not come from one single domain, but actually from many domains. Thus, we define our first research question as the following:

1. Research Question 1.
   What are the research domains with techniques that are applicable to metamodel instance generation?

By identifying each specific research domain, we can define our second research question much more closely with regard to the approaches and techniques within these domains. We define our second research question as:

2. Research Question 2.
   Within those research domains, what theoretical frameworks and associated algorithms have actually been used for metamodel instance generation?

Based on research question 2, we now can define research questions 3 and 4 about the criteria and tools used in the theoretical frameworks and algorithms. Thus, we define the rest of the research questions as follows:

3. Research Question 3.
   What criteria are applied for selecting metamodel instances?

4. Research Question 4.
   What tools exist to implement those algorithms to produce metamodel instances?

We are particularly interested in the criteria for selecting instances because we consider this is an important feature in evaluating a technique for generating metamodel instances. Users may wish to generate a special kind of instance to test their metamodels based on a predefined criteria. In order to generate such a customised instance, it requires users to take good control of the instance generation process.

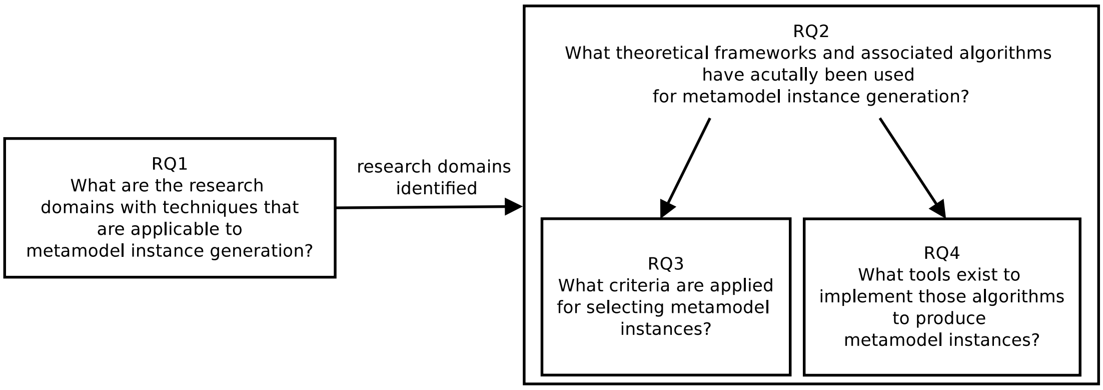

RQ1
What are the research domains with techniques that are applicable to metamodel instance generation?
research domains identified
RQ2
What theoretical frameworks and associated algorithms have acutally been used for metamodel instance generation?
RQ3
What criteria are applied for selecting metamodel instances?
RQ4
What tools exist to implement those algorithms to produce metamodel instances?

Figure 2.1: The relationship between our research questions.

Figure 2.1 shows the relationship between our four research questions. Firstly, by identifying each specific research domain helps us to concentrate on studying approaches and techniques used for generating metamodel instances within those domains. Secondly, research papers that describe theoretical frameworks and algorithms within the identified research domains need to be gathered and reviewed. Finally, the specific criteria and tools that support the theoretical frameworks and algorithms are also studied.

2.2.1 Search terms

In this section, we show that our search terms are derived from two aspects of a metamodel: its structural representations and its applications.

In terms of a metamodel's structural representations, we focus on the different representations that a metamodel could be. For example, a metamodel can be represented as a UML class diagram, or viewed as an abstract syntax tree for a programming language [Heidenreich et al., 2010a; Object Management Group, 2011a,c]. Other types of model which are outside of the scope of this work include state machines and sequence diagrams [Briand et al., 2010; Dinh-Trong et al., 2006; Mahdian et al., 2009; Nayak and Samanta, 2009; Pilskalnsa et al., 2007; Samuel and Joseph, 2008]. To conform to the MOF standard, metamodels discussed in this thesis possess some of the features that a UML class diagram can have. Thus, a metamodel must include one or many of the following features:

1. Classes
2. Attributes
3. Relationships (Generalisations and Associations) between classes

For example, the metamodel (at level 2) in Figure 1.1 depicts two classes (*Class* and *Attributes*) where the association relationship *has* indicates that each *class* can have multiple *attributes*.

In terms of metamodel's applications, we are more interested in the usage of generated metamodel instances. One of the direct applications is to use generated instances for testing model transformations because it requires a large number of instances as test cases. Another application is that a metamodel can also be used for reasoning about an aspect of a system. This is very close to verifying a system's specification. Thus, we also allow our search terms to cover the research from model reasoning, for example: model checking.

The search terms are generated accordingly from these two aspects. We have listed these search terms below:

1. (metamodel test case OR data generation OR metamodel instance generating OR generation OR model transformation testing OR model measurement)
2. (input test models OR automatic model generation OR generating)
3. (grammar testing OR grammar instance generation OR grammar sentence generation OR attribute grammar testing)
4. (XML schema testing)
5. (graph grammar instance OR graph grammar metamodel OR graph grammar testing)
6. (model reasoning OR constraint programming OR model verification OR model checking)

First, some of the terms are directly derived from a metamodel's representation such as "XML schema testing". Second, terms like "model transformation testing" and "model measurement" are derived from applications of generated metamodel instances. Other terms are generated for specific research domains. For example, the term "grammar testing" is formed because a metamodel can convey the meaning of an abstract syntax tree. Also, the search term "model verification" allows us to cover the possibility that users may verify an aspect of a system via verifying models.

The search process is divided into two phases. In the first phase, the search terms are grouped and input into search engines provided by the following databases:

1. Google Scholar
2. ACM Digital Library
3. IEEE Explore
4. ScienceDirect

We reduce the number of papers by looking at the title and abstract of each paper. In the second phase of the search, we review the citations from the results we get in the first phase for any relevant articles, and we repeat this process until no new papers are identified.

In addition, in order not to overlook any papers, we also *manually* search through related conferences and journals for a specific time period (listed in Table 2.1 and Table 2.2). This is because the research on metamodel instance generation actually overlaps many research domains of software engineering. However, not every conference and journal we have searched are certain to contain papers that are related to metamodel instance generation, but to be thorough they are included. We pick these conferences and journals because we consider that some of them are the flagships in each different research domain of software engineering based on their rankings in the field [1], and the others are peer recommended. One may choose differently based on a different ranking engine but we believe Table

[1] We use microsoft academic search engine (http://academic.research.microsoft.com) to check the ranking of each conference and journal in their field.

2.1 and Table 2.2 give us a good list for catching additional key papers. We list these conferences and journals in Table 2.1 and Table 2.2 as additional evidence to support our literature review. For each conference, we widen the time period from the first proceedings of that conference held to the latest proceedings. For each journal, we use the search engine provided within each journal and search through the entire journal.

2.2.2 Paper Selection Study

By judging the titles and abstracts, and studying the related work section for each paper, a total of 43 papers from the search results were finally chosen. We categorise them into Table 2.3 based on 7 different research domains and discuss them in the following sections.

2.3 Discussion

In this section, we provide answers to our each research question defined in section 2.2 and discuss the advantages and disadvantages of each key technique for generating metamodel instances. At the end of this section, we also identify the existing research gaps via outlining the main advantages and disadvantages of each important research domain.

2.3.1 RQ1. What are the research domains with techniques that are applicable to metamodel instance generation?

In this literature review, we are interested in approaches for automatically generating instances (models) from a metamodel. Answering the first research question helps us to identify the major research domains concerning metamodel instance generation.

Name of the Conferences	**Time Period Searched**
Model Driven Engineering Languages and Systems (MODELS)	2005 - 2013
International Conference on Fundamental Approaches to Software Engineering (FASE)	1998 - 2013
International Conference on Automated Software Engineering (ASE)	1997 - 2013
International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)	1995 - 2013
International Conference on Formal Methods (FM)	1999 - 2013
International Conference on Software Testing Verification and Validation (ICST)	2008 - 2013
International Conference on Tests and Proofs (ICTP)	2007 - 2013
International Conference on Software Engineering (ICSE)	1968 - 2013
International Conference on Computer Aided Verification (CAV)	1999 - 2013
International Conference on Model-Driven Engineering and Software Development	2012 - 2013
International Conference on Testing Software and Systems (ICTSS)	2010 - 2013
European Conference on Modelling Foundations and Applications (ECMFA)	2005 - 2013
International Conference on Model-Driven Engineering and Software Development (MODELSWARD)	2013 - 2013
International Symposium on Foundations of Software Engineering (FSE)	1993 - 2013

Table 2.1: Conference proceedings examined

Name of the Journal
ACM Transactions on Software Engineering and Methodology
Software: Practice & Experience
IEEE Transactions on Software Engineering
Empirical Software Engineering
Science of Computer Programming
Software and Systems Modeling
Journal of Object Technology
Information and Software Technology
Automated Software Engineering
Software Testing, Verification & Reliability
Journal of Systems and Software

Table 2.2: Journals examined

2.3.1.1 Compiler Testing

A computer program can be syntactically parsed into a parse tree, and a parse tree is a graphical view of an instance of a programming language grammar. In other words, a syntactically correct computer program must conform to its language's grammar. This is similar to the concept of a sample model instance that conforms to a metamodel. Thus, producing a sample model from its metamodel is related to the problem of generating programs from a given grammar, which is in the domain of compiler testing [Boujarwah and Saleh, 1997].

2.3.1.2 Model Transformation Testing

Model transformation testing is also related to instance generation. Model transformations are the essential feature of Model-Driven Engineering (MDE). The process of model transformation takes as input a model which conforms to a source metamodel and transforms it into another model which conforms to a different target metamodel. There are two ways to test this process. One way is testing the model transformation program code itself [Küster and Abd-El-Razik, 2006]. For example, making sure that all the statements are executed at least once. This is a white-box testing technique and it requires testers to have knowledge of the internal logic or the code structure of the programs. The other way is to produce a large set of sample models and to run the transfor-

References	**Research Domain**
[Alanen and Porres, 2003]	Compiler testing
[Brottier et al., 2006; Fleurey et al., 2004, 2009; Lamari, 2007; Sen and Baudry, 2006]	Model transformation testing
[Ehrig et al., 2009; Heckel and Mariani, 2005; Hoffmann and Minas, 2011; Winkelmann et al., 2008]	Graph grammar
[Anastasakis et al., 2010; Bordbar and Anastasakis, 2005; Jackson, 2002; Torlak and Jackson, 2006, 2007], [Anastasakis et al., 2007; Sen et al., 2008, 2009; Soeken et al., 2011a], [Büttner and Cabot, 2012; Clavel et al., 2009; Wille et al., 2012; Yatake and Aoki, 2012], [Garis et al., 2011; Jackson et al., 2011; Kuhlmann and Gogolla, 2012; Kuhlmann et al., 2011; McQuillan and Power, 2008; Soeken et al., 2010, 2011b]	SAT/SMT based approaches
[Cabot et al., 2007, 2008, 2009; Cadoli et al., 2004, 2007; González Pérez et al., 2012]	Constraint Programming approaches (CP)
[Bertolino et al., 2007a,b]	XML
[Balaban and Maraee, 2013; El Ghazi and Taghdiri, 2011; Gogolla et al., 2005; Lukman et al., 2010; Mougenot et al., 2009; Queralt et al., 2012]	Miscellaneous Domains

Table 2.3: Papers are grouped by 7 research domains: compiler testing, model transformation testing, graph grammar, SAT/SMT based approaches, Constraint Programming, XML and other.

mation program with them to check the correctness of the results regardless of the internal structure of the transformation program. This is a black-box testing technique since the testers do not have knowledge of the code at all [Beizer, 1995]. The most recent work on testing model transformation is by Macedo et al. [Macedo and Cunha, 2013], who encode bidirectional model transformation QVT-R semantics (Query/View/Transformation Relation) into Alloy (a formal specification language that supports automatic SAT solving) to perform semantic checking [Object Management Group, 2011b]. Their approach supports the encoding of metamodels annotated with OCL constraints in the Alloy specification language. Therefore, the research that has been done to automatically produce sample models in this domain is closely related to our interests.

2.3.1.3 Graph Grammars

Another research domain we are interested in is graph grammars (GG). Graph grammars provide a theoretical foundation for graphical languages [Rozenberg, 1997]. A graph grammar consists of a set of rules where each rule can be applied if there are any sub-graphs that match. For example, Figure 2.2 shows an example of a grammar rule. This rule indicates that if a graph or sub-graph matches with the pattern on the left hand side (LHS) in Figure 2.2, the right hand side rule (RHS) applies. Thus, an edge going from node A to node B is established for the graph on the LHS. This idea is derived from the concept of a context-free grammar (CFG), except that a CFG deals with strings while a GG is associated with graphs. Graph grammars can be used to specify a software model and they provide a natural way of generating instances [Hoffmann and Minas, 2010]. Similarly, a metamodel graphically describes models and can be considered as a graph in the sense of graph grammars. Therefore, the natural derivation process of a graph grammar can be adapted for generating metamodel instances. Thus, the techniques that have been developed in this domain to generate sample models are also included in this literature review.

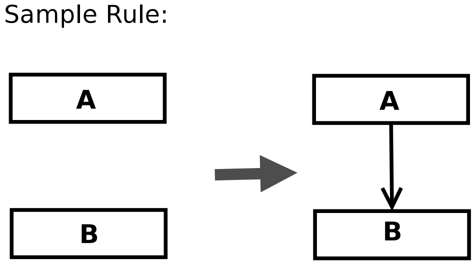


Figure 2.2: A sample rule of a graph grammar

2.3.1.4 SAT/SMT based Approaches

SAT/ Satisfiability Modulo Theories (SMT) solvers are often used to reason about the consistency of a model [Armando et al., 2009; Chang, 2007; Cordeiro and Fischer, 2011]. Research has demonstrated that models can be encoded as boolean logic formulas. The resulting formulas are transformed into Conjunctive Normal Form (CNF) via a standard transformation, and the final CNFs are solved by a SAT solver [Jackson, 2000; Tseitin, 1968]. Each successful assignment for the boolean logic formulas is then translated back into the problem domain. The major difference between SAT and SMT solvers is that SMT solvers are supported by rich background theories such as linear integer arithmetic theory, while SAT solvers only use propositional logic [Barrett et al., 2010]. Using SAT/SMT solvers, a model's properties can be verified within a limited search space by finding an instance. Since a metamodel is a model that describes other models, the research that employs SAT/SMT solvers to verify model properties is highly relevant.

2.3.1.5 Constraint Programming

Similar to SAT/SMT solvers, constraint programming is another important technique that can be used for solving constraint problems. A model or specification can be rewritten as a constraint problem in constraint logical languages. The problem can be further examined by using different libraries or platforms that are provided by constraint logical languages. For example, ECL^{i}PS^e provides users with a platform for writing constraint problems and solving them [Apt and Wallace, 2007]. The difference between constraint programming and SAT/SMT solvers is that constraint programming allows problems to be programmable, while SAT/SMT solvers are used as a black-box engine [Bordeaux et al., 2006].

2.3.1.6 XML & Miscellaneous Domains

Extensible Markup Language (XML) is a simple text format both readable by machines and humans [World Wide Web Consortium, 2008]. A MOF metamodel is defined in the format of XML Metadata Interchange (XMI) which is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML) [Object Management Group, 2011a]. Metamodel instances can possibly be achieved via generating XML instances. Therefore, the work done in this research domain is also related to metamodel instance generation. We categories other techniques that can also be used for generating metamodel instances into Miscellaneous Domains in Table 2.3, and discuss them in section 2.3.2.7.

2.3.2 RQ2. Within those research domains, what theoretical frameworks and associated algorithms have actually been used for metamodel instance generation?

In this subsection, we group metamodel instance generation techniques into 7 research domains. These 7 domains (given in Table 2.3) are compiler testing, model transformation testing, graph grammars, SAT/SMT based approaches, constraint programming approaches (CP), XML and miscellaneous domains. We examine the papers that appear in each research domain, and discuss the capabilities of their theoretical frameworks and algorithms.

2.3.2.1 Compiler Testing

In the domain of compiler testing, Boujarwah and Saleh [1997] present a survey about existing compiler testing techniques. Most of the techniques in their survey use context free grammars (CFG) to generate test strings. For example, a sentence generation algorithm generates sentences from a CFG [Purdom, 1972]. In order to generate semantically correct programs, Harm and Lämmel

[2000] propose a framework that generates programs using an attributed grammar. However, this framework only works for small scale grammars and no evidence indicates that it can be applied to the general case. Much research on grammar testing has already shown a method for direct test case generation from a given language grammar [Lämmel, 2001; Lämmel and Schulte, 2006]. However, instance generation is more complicated than the derivation tree of a sentence. This is because the generation process must take care of additional constraints defined on a metamodel. For example, a valid instance of a metamodel conforms to the additional constraints defined for an attribute of a class in a metamodel. However, the process of constructing a derivation tree for a sentence does not take these constraints into account. Therefore, the techniques used for testing a compiler can not be directly applied to metamodel instance generation.

Alanen and Porres [2003] propose two algorithms, one to transform a metamodel to a context-free grammar, and the other one to derive a metamodel from a context-free grammar. Their algorithm for the derivation of a grammar from a metamodel is limited with respect to the structure of the metamodel, as it can only deal with composite associations in the metamodel. By using their algorithm to transform a metamodel into a grammar, it is possible to apply an existing string generation algorithm to get sample strings from the grammar, and then translate the sample strings back into instances of the metamodel. However, if we define a metamodel that preserves properties of a UML class diagram, it is not surprising that their algorithms cannot deal with multiplicities of an association.

2.3.2.2 Model Transformation Testing

In the model transformation testing research domain, some techniques for automatic model instance generation have been developed. Brottier et al. [2006] propose an algorithm to automatically generate a set of model instances for testing a model transformation. In this algorithm, the authors use two concepts: model fragments and object fragments. To form an object fragment, they employ the classic category-partition testing technique [Ostrand and Balcer, 1988]. They partition the metamodel into different equivalence classes by using the coverage criteria for testing the UML class diagram [Andrews et al., 2003]. For example, an integer type attribute P of a class C is partitioned into three categories $\{P = 0, P < 0, P > 0\}$. Each element in the set is marked as an object fragment, and an object fragment specifies a partial instance of the class C. A model fragment consists of a list of such object fragments. The algorithm takes in a set of model fragments and outputs a set of models that conform to the metamodel.

The algorithm of Brottier et al. [2006] iterates over the uncovered model fragments and keeps generating model instances until all the model fragments are covered. Furthermore, this algorithm provides some strategies for testers to meet specific testing needs. For example, it provides a strategy that reuses generated model instances as much as possible to achieve the smallest number of instances to cover most of the model fragments. However, this algorithm has two main limitations. One is that it supposes that all the model fragments are manually provided by testers. The other is it does not take Object Constraint Language (OCL) constraints into account [Warmer and Kleppe, 1999]. Therefore, the constraints defined in the metamodel are overlooked and, as a result, the generated model instances do not satisfy these constraints.

Baudry et al. [2002b] propose a bacteriologic algorithm which is based on genetic algorithms. This algorithm consists of 4 steps. Before executing the algorithm, an initial set of the test models has to be provided. In the first step of the algorithm, the test models are ranked using a fitness function which estimates the level of the coverage that they can make. In step 2, the test models which make the most significant contribution to coverage are recorded and added to the solution set. The test models which can not make any contribution to the coverage are removed in step 3. In the final step, a mutation operator is applied to the best test models in order to create new test models. The algorithm keeps iterating until all coverage items are covered by the test models. Baudry et al. [2002a] implemented this algorithm. However, the limitation of this algorithm is that it requires an initial set of test models. Therefore, to compute such a set of test models is another problem of this approach.

2.3.2.3 Graph Grammars

We have also studied research on graph grammars. Ehrig et al. [2009] extend the work by Bardohl et al. [2004], and propose an instance-generating graph grammar for creating metamodel instances. In their approach, they use an *attributed type graph* [2] to capture metamodel structures, and the concept of layered graph grammars to order rule applications. Metamodels are represented as attribute type graphs, and each rule is applied when a specific metamodel pattern is found in the attribute type graph and the corresponding application conditions are satisfied. Three layers are defined for the rules. The rules in layer 1 are used for creating instances of each non-abstract class in the metamodel. Layer 2 consists of a set of rules that deal with three metamodel patterns. These patterns are those association relationships defined in the metamodel with a multiplicity of one. The rules in layer 3 are defined to address associations with multiplicity of zero to n. The authors illustrated this approach by applying these rules to an example.

However, in the work of Ehrig et al. [2009] composition associations are not considered. This was later tackled by Hoffmann and Minas [2011]. Their work uses graph grammars, called adaptive star grammars, to show how a metamodel (represented as a class diagram) can be translated into this grammar. Such a graph grammar can be used to automatically generate instances of the metamodel [Drewes et al., 2006]. In this approach, a set of adaptive star grammar rules are introduced to deal with relationships defined in the metamodel. These include unique, non-unique and composition associations. However, OCL constraints defined over the class diagram can not be handled by this approach.

To deal with OCL constraints, Winkelmann et al. [2008] present a method in which OCL constraints can be translated into graph constraints. In their approach, they show how a list of OCL constraints can be translated into equivalent graph constraints. These OCL constraints are restricted to equality, size and attribute operations for navigation expressions, called restricted OCL constraints. They suggest two ways of generating a valid model instance that satisfies the OCL constraints. One way is to check the translated constraints after an instance is

[2] We formally describe it in Chapter 3

generated. The other way is by taking the constraints into consideration during the instance model generation process. Both ways have their advantages and disadvantages. The former may lead to a large number of invalid model instances being generated more quickly, while the later produces only those model instances which satisfy the constraints, but are generated more slowly.

2.3.2.4 SAT/SMT Based Approaches

SAT/SMT based approaches are popular in the area of formal verification, and many research have been built upon utilising SAT/SMT solvers to verify programs or models [Armando et al., 2009; Cordeiro and Fischer, 2011; El Ghazi and Taghdiri, 2011; Tianhai Liu, 2012]. Among them Alloy is one of the most popular research that uses SAT solvers [Jackson, 2002]. Alloy, a model finder, is a well designed tool that can be used for finding instances of a model and counter-examples in a finite search scope [Chang, 2007; Jackson, 2002, 2006; Jackson et al., 1998, 2001]. Alloy, unlike other model-finding tools such as MACE [Claessen and Sörensson, 2003], uses first-order relational logic to describe a model. To speed up the searching process and guarantee termination, Alloy bounds relational logic, translates the bounded relational logic into a boolean formula, and then it calls an external SAT solver to solve the boolean formula. The solutions returned by the SAT solver then get translated back into the models [Torlak and Jackson, 2006, 2007; Torlak et al., 2008; Torlak, 2009].

Since Alloy can be used to generate instances of a model within a bounded search space, research with Alloy has been highly active [Anastasakis et al., 2007, 2010; Bordbar and Anastasakis, 2005; Garis et al., 2011; Kuhlmann and Gogolla, 2012; McQuillan and Power, 2008; Sen et al., 2009; Shah et al., 2009]. Anastasakis et al. [2007] focus on transformation between UML class diagrams and the Alloy language. In their work, they demonstrate a list of rules which can map a UML class diagram and a limited number of OCL constraints to the Alloy language. Sen et al. [2008] are inspired by this approach and present a tool called Cartier to automatically generate test models for testing model transformations. This tool can transform a metamodel (in Ecore format) with OCL constraints, model transformation pre-conditions, model fragments and test objectives into the Alloy language [Fleurey et al., 2009]. They also present some strategies based on partition techniques to guide the tool to generate models [Sen et al., 2009]. These strategies are written as Alloy predicates, combined with models and are solved by SAT solvers. The solutions returned by SAT solvers are then transformed by the Cartier tool back to the instances of the input metamodel.

Another related approach is by McQuillan and Power [2008]. They propose a metric metamodel for the measurement of object-oriented software. In their work, they firstly transform the metrics metamodel along with the OCL constraints into an Alloy specification, and then use Alloy to generate possible instances. To transform the Alloy generated instances back to the metamodel instances, they developed a tool called Reflective Instantiator. This tool reads an Alloy specification file and creates an instantiation of the Java implementation of the metamodel.

Both the approaches of Sen et al. [2008] and McQuillan and Power [2008] translate a metamodel and OCL constraints to Alloy. However, the problem for both approaches is that any OCL constraints that involve numbers cause kodkod (Alloy's engine) bit-blast [Torlak, 2009]. This significantly slows down the translation process. The reason for this is that kodkod is based on SAT-solving, and SAT solvers are not designed to solve numeric constraints. Thus, numeric constraints are beyond the capability of kodkod. Another difficulty is that both approaches utilise Alloy's APIs to invoke the model generation process, thus both tools are not easy to maintain since they are highly dependent on Alloy.

Kuhlmann and Gogolla [2012] propose a way to translate OCL collection data types into Alloy's first-order relational logic. This translation is based on a uniform representation that they describe in the work. Each collection data type (Set and Bag) is flattened into a relation which is later translated into boolean formulas via Alloy's engine: kodkod. Each successful assignment for these boolean formulas is converted back into actual values that are contained by each collection data type. However, their experimental results show that this approach could only be applied to manually designed examples. Furthermore, any numerical constraints involved in the collection data types are handled poorly. This is because kodkod is a SAT-based engine for Alloy, and SAT solvers are not designed particularly well for solving numerical constraints.

Many SMT-solvers are well developed and supported by multiple theories, such as lists, sets etc [Bruttomesso et al., 2008, 2010; De Moura and Bjørner, 2008]. Compared to SAT solvers, SMT-Solvers have a greater advantage because of the multiple theories defined in the Satisfiability Modulo Theories Library (SMT-Lib) [Barrett et al., 2010]. Unlike SAT solvers whose input is in Conjunctive Normal Form (CNF), SMT-solvers are formulated according to a specific theory. A typical example is solving a linear integer arithmetic equation, where the input is a set of equations written in human readable format and the output is the assignment for each variable in the equations. Thus, we have also studied approaches that take advantage of SMT-solvers to find instances of a model [Jackson et al., 2011; Soeken et al., 2010, 2011b].

Soeken et al. [2010] encode a UML class diagram as a set of operations on bit-vectors which can be solved by SMT solvers using bit-vector theory. A successful assignment for each bit-vector is interpreted as an instance of a UML class diagram. Soeken et al. [2011b] propose an approach to encode a subset of OCL constraints as bit-vectors, and provide a list of the corresponding mappings between OCL collection data types (Set, Bag, Sequence) and bit-vector operations. Furthermore, allocating an appropriate bound for each entity defined in the model is essential for bounded model checking and verification [Soeken et al., 2011a]. Soeken et al. [2011a] propose a linear integer arithmetic approach. This approach considers different kinds of associations defined in the model, translates defined multiplicities in each association into linear integer arithmetic equations, and uses an SMT solver to solve the equations. The bound for each entity can therefore be determined by solved equations. However, this approach cannot deal with OCL invariants that specify the number of instances to be created.

Jackson et al. [2011] propose a framework called Formula, which is a MOF-like framework that can capture metamodeling's abstraction via a graph-like language. This framework consists of three components: a model store to specify models and metamodels, a list of operations to edit models and metamodels, and a meta-interpreter which can promote model-level elements to meta-level elements. To be able to verify properties of operations on models and metamodels, the generated constraints are solved by an SMT-Solver [De Moura and Bjørner, 2008]. However, their framework is different from standard metamodeling approach. Thus, a conversion from standard metamodels to their framework is necessary. This becomes almost infeasible if there are a large number of metamodels because Formula itself does not provide any tools that can automatically convert metamodels into their framework.

We have also considered other work that takes advantage of SMT-Solvers to complement Alloy techniques. El Ghazi and Taghdiri [2011] present work that translates an Alloy specification into SMT-instances to prove Alloy assertions. This work also takes advantage of linear integer arithmetic which is supported by an SMT-solver to perform unbounded integer arithmetic operations.

2.3.2.5 Constraint Programming Approach (CP)

One of the earliest works on finite model reasoning uses description logics to encode a UML class diagram [Calvanese, 1996]. Cadoli et al. [2004] use this idea to implement a technique that can encode a UML class diagram into linear inequalities that can be solved by a constraint programming solver [Calvanese, 1996; Calvanese and Lenzerini, 1994; ILOG, 2001, 2002]. They can construct a model based on solved inequalities which provides information regarding the cardinalities of instances of each class. More specifically, they use a boolean array to represent a finite universe and the size of the universe is determined by the solution to the inequalities. However, this work does not consider any OCL constraints, it also requires that users have significant knowledge on how to encode a UML class diagram into inequalities, and the tool fails to demonstrate any possibility for automation.

Cabot et al. [2008] propose a procedure that can transform a UML class diagram with OCL constraints into a Constraint Satisfaction Problem (CP) according to a set of rules. The CP is described using the syntax provided by the ECL^{i}PS^{e} Constraint Programming System [Apt and Wallace, 2007]. The CP itself is divided into two subproblems. The first subproblem is to determine a bound for each class and association (variables), and the second is to assign a value to each variable. If CP has a solution (instance), the user can conclude that a model satisfies the properties. Cabot et al. [2009] extend their work to OCL operation contracts by using a similar translation process to the one used in their earlier work [Cabot et al., 2008]. Their work is supported by a tool called UMLtoCSP, and a tool called EMFtoCSP which extends UMLtoCSP to deal with EMF metamodels [Cabot et al., 2007; González Pérez et al., 2012]. Both tools are designed for bounded verification and they do not provide an instance enumeration mechanism like Alloy.

2.3.2.6 XML

Bertolino et al. [2007a] propose an approach called XML-based Partition Testing (XPT) to automatically generate XML instances from a XML Schema. This approach uses the idea of a classic Category Partition method to derive a set of XML instances from a preprocessed XML Schema [Ostrand and Balcer, 1988]. The XPT methodology consists of five steps and is partially implemented in a proof-of-concept tool called "Testing by Automatically generated XML Instances (TAXI)" [Bertolino et al., 2007b]. In the first step, a XML Schema is preprocessed and some of the elements in the XML Schema are rewritten in order to facilitate the later steps. Next, an occurrence analysis is adopted to manipulate the boundary values for the occurrence of each element in the XML Schema. The third step is to assign a value to each element. These values are either randomly generated or selected from a database. A set of intermediate instances are then generated by combining the occurrences of each element in the fourth step. Finally, the actual XML instances are generated from the intermediate instances. However, this approach does not handle the associations defined in the model and furthermore it does not consider any OCL constraints defined in the model.

2.3.2.7 Miscellaneous Domains

Gogolla et al. [2005] propose an approach that uses the language ASSL. ASSL is an extension language of the USE specification language [Gogolla et al., 2007]. It provides a list of commands for testers to generate snapshots for finding defects in a UML class diagram. A snapshot is an object diagram that represents the system states at any time with objects, attribute values and links. The snapshots can be created by an ASSL procedure, and moreover the snapshots can be created with consideration for OCL invariants. With this approach, it is possible to generate a number of snapshots to validate the design at an early stage. However, one must know how to write an efficient ASSL procedure in order to generate appropriate snapshots.

Mougenot et al. [2009] use an approach that is based on the Boltzmann method to generate metamodel instances [Duchon et al., 2004]. In their approach, a metamodel is first transformed into a Boltzmann tree specification and the final instances are derived from the generated trees. The main purpose of this approach is to make sure that the instance generation process has no bias, and therefore the generated instances are uniform. Another feature of this approach is that the Boltzmann method has a linear property, which means that the instance generating process is linear with respect to the size of the generated instances. However, this approach does not take into account any constraints defined in the metamodel. Hence, instances that do not satisfy constraints can be generated.

Lukman et al. [2010] propose a traversal algorithm for metamodels. In their algorithm, the root element of a metamodel is first identified. A root element in a metamodel is a container class that can be used to form a composition relationship with other classes. This algorithm traverses each class connected by association, generalisation and composition relationships that are defined in the metamodel, and marks the paths visited to prevent an infinite iteration of each class. Although their work does not describe how to generate a model instance, by applying this algorithm, it is possible to generate an instance from each class that has been traversed. However, traversing a metamodel is not sufficient to produce a valid instance from the metamodel. For example, a valid instance must meet the multiplicities of an association, which requires that the algorithm be aware of the links between objects.

Lastly, Balaban and Maraee [2013] propose an algorithm (called FiniteSat) to check the satisfiability of a UML class diagram. This algorithm is particularly concerned with class hierarchy constraints such as disjoint and complete constraints defined in an inheritance relationship. A disjoint and complete constraint means that an instance of a superclass is one of its subclasses and each instance of a subclass have no members in common [Object Management Group, 2011d]. Their algorithm is also concerned with the multiplicities defined on an association. The instances of two classes between an association must satisfy their corresponding multiplicities. Their algorithm returns an answer to indicate whether a UML class diagram is consistent or not. However, their algorithm does not take OCL constraints into account, and it does not generate any instances. Thus, their approach can only be applied to check the consistency of a metamodel rather than instance generation.

2.3.3 RQ3. What criteria are applied for selecting metamodel instances?

Fleurey et al. [2004] propose metamodel coverage criteria. These originate from existing coverage criteria that have been adapted from UML class diagrams. Since a MOF metamodel is similar to a UML class diagram, they reuse existing UML coverage criteria for metamodels. These coverage criteria were first proposed by Andrews et al. [2003] and are defined for a UML class diagram using three criteria:

1. Association-end multiplicities (AEM): Each representative link in a class diagram must be covered.
2. Class attribute (CA): In each instantiated class, the set of representative attribute value combinations in each instance of a class must be covered.
3. Generalisation (GN): Every specialisation defined in a generalisation relationship of a class diagram must be covered.

Two of the criteria listed above (AEM and CA) use representative values that are based on partition testing techniques to express their meanings. The representative values can be selected by applying knowledge-based or default partitioning approaches. For example, an integer attribute can be partitioned into three values which are greater, less than and equal to zero. For knowledge-based partitioning, the values can be provided by testers or determined by the specification of the model, e.g. by examining the constraints defined over the model. For default-partitioning, the representative value can be selected by partitioning an attribute into minimum, non-boundary and maximum values. Fleurey et al. [2004] adapt these two criteria to achieve metamodel coverage and define their test criterion. Later this criterion was implemented in the work [Brottier et al., 2006].

To continue the work in Brottier et al. [2006], Fleurey et al. [2009] proposed an approach to select model fragments, a list of test criteria and a tool developed for automatically generating model fragments. They also designed a generic test criteria metamodel, where the generated model fragments from the tool must conform to this metamodel. To evaluate whether a set of test models satisfy the criteria defined by the test criteria metamodel, they use a tool called MMCC which involves four steps. In the first and second step, the tool automatically generates a set of model fragments according to the test criteria. Then in step 3 these generated model fragments are checked against the test models. The final step shows uncovered model fragments, if there are any, and requires a manual analysis of the reasons why those uncovered model fragments remain. The test criteria defined in their approach are defined using OCL constraints over the test criteria metamodel. They propose 6 criteria, which are shown in Table 2.4.

Among the 6 criteria listed in Table 2.4, those criteria marked with the type "object fragment" can be combined with those marked as a "model fragment" to form the test criteria. For example, the combination of criteria 3 and 5 results in one model fragment that contains all possible combinations of ranges of the properties of a class. However, the criteria defined in their approach produce quite a high number of object fragments since they use a Cartesian product on ranges. This may lead to a combinatorial explosion if the input metamodel has a large number of properties defined in each class.

2.3.4 RQ4. What tools exist to implement those algorithms to produce model instances?

A number of tools have been developed for producing or assisting automatic instance generation. The tools that support model instance generation can be mainly grouped into two categories: independent and dependent. An independent tool can be run without the support of any other tools or platforms, while a dependent tool requires some support. The tools are given in Table 2.5 with a column to indicate their category.

As mentioned already in section 2.3.2.4, Alloy is a popular tool used by many modelers to find instances or counter examples of a model. It was introduced

Name of the criteria	Description	Type
1. AllRanges	All ranges of a property of a class to be covered.	N/A
2. AllPartitions	All ranges of a property of a class to be covered in one test model.	N/A
3. OneRangeCombination	Every range of each property of a class has to be covered at least once.	Object Fragment
4. AllRangeCombination	One object fragment must contain all possible combinations of ranges of a property of a class.	Object Fragment
5. OneMFPerClass	A single model fragment must contain all possible combinations of ranges for a class.	Model Fragment
6. OneMFPerCombination	Every model fragment only contains one possible combination of ranges for a single class.	Model Fragment

Table 2.4: Six test criteria defined in Fleurey et al. (2009)

Name of the tool and Reference	Category
Alloy [Jackson, 1998, 2000, 2002; Torlak and Jackson, 2007]	Independent
Formula [Jackson et al., 2011]	
OMOGEN [Brottier et al., 2006],	
MMCC [Fleurey et al., 2009],	
Reflective instantiator [McQuillan and Power, 2008],	
USE [Gogolla et al., 2005]	
Echo [Macedo and Cunha, 2013] UML2Alloy [Anastasakis et al., 2007; Bordbar and Anastasakis, 2005]	Depends on Alloy
Cartier [Sen et al., 2009],	Depends on Alloy
USE model Validator plugin [Kuhlmann et al., 2011],	Depends on kodkod
UMLtoCSP, EMFtoCSP [Cabot et al., 2007; González Pérez et al., 2012],	Depends on ECL ^{i}PS^{e}
A generator as an Eclipse Plugin [Mougenot et al., 2009]	Depends on Eclipse

Table 2.5: Tools that support or assist automatic model instance generation

in [Jackson et al., 2000] and it uses a relational logic to capture the semantics of first-order logic, and thus a model is described in a relational logic [Jackson, 1998]. The later version is improved so that it could allow modelers to specify quantifiers over the relational logic [Jackson, 2002]. The latest version of Alloy employs a new engine (kodkod) that represents a model as a bounded relational logic, then translates it into a boolean formula that is represented by compact boolean circuits [Torlak and Jackson, 2007; Torlak, 2009]. This finally solves the formula by using external SAT-solvers [Berre and Parrain, 2010; Een and Sörensson, 2005; Moskewicz et al., 2001]. With the kodkod engine, Alloy is able to deal with larger models, and find instances or counter examples in seconds. However, one of the main problems with using Alloy to find model instances is that it performs poorly in solving numeric constraints. This is because kodkod uses SAT solvers as its solving engine and SAT solvers are not capable of solving numeric constraints. Furthermore, it is difficult for users of Alloy to generate customised instances. For example, generating instances that meet pre-defined criteria. This is because in order to be able to generate such instances, users must gain full control of generation process and Alloy does not grant such control.

USE is a tool that can be used to build a variety of UML diagrams such as UML class diagrams and sequence diagrams [Gogolla et al., 2007]. It also provides the user with a way of creating and checking OCL constraints defined over the model, and fully supports OCL 2.0 [Object Management Group, 2012a]. The latest research for USE involves integrating it with Alloy's core engine [Kuhlmann et al., 2011]. The resulting integration with kodkod yields a USE Model Validator plugin. This plugin provides users with an interface that allows a user to specify a lower and upper bound for each class defined in the UML model. Since this approach is fully dependent on the performance of kodkod, it thus requires a translation from UML class diagrams and OCL constraints into relational logic, which complicates the instance generation process.

Brottier et al. [2006] implement their algorithm in a tool called OMOGEN. This is a prototype tool and it can be used to generate model instances based on the input of a set of model fragments. However, to generate the model fragments a tool called MMCC is needed [Fleurey et al., 2009]. This tool is implemented using the Eclipse Modeling framework (EMF). It generates partitions from a source metamodel and model fragments according to particular test criteria which were described in Table 2.4.

The reflective instantiator described in McQuillan and Power [2008], parses generated instances from Alloy into instances of the Java implementation of the metamodel. Furthermore, this tool generates a code implementation of a metamodel.

Gogolla et al. [2005] extend USE features by defining a new language to generate snapshots of the models. These snapshots are treated as test cases and can be validated within USE. Their approach allows OCL invariants to be dynamically loaded and certified during execution. However, it does not provide any test criteria during the generation of snapshots and any test criteria have to be manually programmed.

The Cartier Tool depends on Alloy [Sen et al., 2009]. It invokes Alloy APIs to launch the SAT solver to generate model instances. However, Cartier does not automatically transform the OCL constraints of the input metamodel into Alloy facts. Furthermore, Cartier requires that a set of model fragments is generated before generating model instances. These model fragments are generated by the tool introduced in [Fleurey et al., 2009]. Therefore, Cartier heavily depends on other tools and this makes the model generation process quite complicated.

The Boltzmann method is implemented in a generator as an Eclipse Plugin [Mougenot et al., 2009]. This generator produces a valid model by identifying properties, generalisation and references in the metamodel. This tool is very effective for generating a large uniform sample of models. However, this tool can not generate models that will satisfy the constraints defined with respect to the metamodel.

Macedo and Cunha [2013] develop a tool called Echo that can be used for verifying model transformation. Echo is based on Alloy's engine kodkod. Thus, it translates a metamodel and OCL constraints into first-order relational logic (Alloy's specification). However, Alloy's specification does not support multiple-inheritance relationships. Therefore, Echo cannot deal with any multiple-inheritance relationships in a metamodel. Furthermore, since kodkod uses a SAT solver as back-end reasoning engine, any numerical constraints in the metamodel can cause kodkod bit-blasting, and slow down the translation.

2.3.5 Discussion

The majority of the research done on instance (model) generation techniques comes from four research domains: model transformation testing, graph grammars, SAT/SMT based approaches and constraint programming, as can be seen in Table 2.3. In order to effectively identify the gap in the metamodel instance generation, we discuss the advantages of disadvantage of these four research domains and propose a new research direction to the problem of metamodel instance generation.

In the model transformation testing research domain, algorithms, frameworks and tools have been developed for generating or assisting metamodel instance generation [Brottier et al., 2006; Fleurey et al., 2004, 2009; Lamari, 2007; Sen and Baudry, 2006]. However, most of them cannot handle OCL constraints with respect to the metamodels. To improve this, Macedo and Cunha [2013] propose the most recent technique in this area. Their approach is based on Alloy's specification. In their work, metamodels and OCL constraints are both transformed into an Alloy specification and subsequently solved by Alloy's solving engine (kodkod). Each successful solving returns a valid instance. However, one big drawback in their approach is that not all metamodel structure features are supported by the Alloy specification. For example, Alloy does not support multiple-inheritance relationships. To allow multiple-inheritance relationships, either users must manually edit all multiple-inheritance relationships into single relationships or tune Alloy's engine to support this feature. However, both ways require a significant amount of work. This is almost infeasible when the size of a metamodel is very big and has many multiple-inheritance relationships. Also, tuning Alloy is not an easy task since it consists of more than $50,000$ lines of code over 400 classes [Chang, 2007]. Another big drawback is that since Alloy uses a SAT solver as its back-end engine, any numeric OCL constraints in the metamodel can cause bit-blasting and significantly slow down the solving time [Torlak, 2009].

SAT/SMT solvers appear to be very successful in solving constraint problems. Constraints are expressed as boolean formulas. Among SAT/SMT approaches, Alloy is the most mature tool in use, with many researchers working with this tool. [McQuillan and Power, 2008; Sen et al., 2008, 2009]. Alloy uses a SAT solver as its back-end, and SAT solvers are designed for solving boolean formulas, not for handling any integer arithmetic reasoning. Any numerical constraint solving is beyond the capabilities of SAT solvers. Therefore, all research using a SAT solver as the back-end suffers this problem [Anastasakis et al., 2007; Bordbar and Anastasakis, 2005; Macedo and Cunha, 2013]. Furthermore, Alloy is not able to generate instances to meet specific coverage criteria in the first place, so a test case reduction technique must be applied to remove unnecessary instances [McQuillan and Power, 2008]. On the other hand, SMT solvers are suitable for solving both pure boolean formulas and numeric constraints in many areas because of their well crafted decision procedures for handling different theories [Armando et al., 2009; Cordeiro et al., 2009; Tianhai Liu, 2012]. However, existing work with SMT solvers in the area either does not support the standard metamodeling approach or provides no supporting automated tools [Jackson et al., 2011; Soeken et al., 2011b].

Similarly, constraint programming (CP) approaches have been adapted for generating metamodel instances. In the work of Cabot et al. [2008], they rewrite the problem of generating UML class diagram instances as a constraint problem. The main advantage is that CP provides a high-level language (close to general-purpose programming languages) so that a particular constraint problem is programmable. CP also offers solvers that can handle both boolean and numeric constraints. However, the disadvantages are that CP is not well-suited to be used as a black-box engine compared to a SAT/SMT approach, making full automation difficult. In terms of measurability of progress, constraint programming is very poor compared to SAT/SMT research community [3] [Bordeaux et al., 2006]. Furthermore, a SAT/SMT approach offers very much on a general purpose constraint-solving method that could be naturally adapted to the problem, while constraint programming requires much more expertise and programming skills for a particular problem. In fact, this is one of the main challenges to improve constraint programming [Puget, 2004].

Another research domain which has been exploited to overcome the disadvantage of metamodeling is that of graph grammars. Graph grammars offer a natural way to describe the derivation process and so have an advantage for generating metamodel instances. The biggest advantage of using graph grammars is that it transforms a metamodel into a type graph, and a type graph naturally captures the structure of a metamodel such as inheritance and association relationships. However, one of the main problems for graph grammars is that graph parsing is very expensive because it is not always deterministic as a rule may match several sub-graphs. Another key issue is that in general determining if a given graph belongs to a particular graph language is *undecidable* [Ehrig et al., 2006]. This problem is well-known as *membership problem* in formal language theory [Poonen, 2012]. For graph grammars this problem is lifted to graphs. Thus, the program may never terminate. The other problem is that current techniques that work with graph grammars do not automatically support selection criteria for metamodel instances. In order to do that, one must manually write out the grammar rules and perform the generation process.

We summarise the main advantages and disadvantages for each research do-

[3] http://www.satcompetition.org

Research Domain	**Advantage**	**Disadvantage**
Model transformation testing	Supports direct application of instance generation	Poor OCL support
SAT	Supports boolean constraints	Poor integer arithmetic support
SMT	Supports boolean and integer arithmetic constraints	No automation or standard metamodeling approach support
Constraint programming	Supports boolean and integer arithmetic constraints	Requires expertise in constraint programming and not suitable to be used as a black-box engine
Graph grammars	Supports metamodel structure	Expensive graph matching and membership problem

Table 2.6: A summary of the advantages and disadvantages of each research domain identified. Note that we separate the research domain SAT/SMT here for the sake of clarity.

main discussed above, and list them in Table 2.6. As can be seen, each research domain has its own advantage and disadvantage on generating metamodel instances. However, in relation to our problem statement described in section 1.6, none of them provides a mechanism that is able to generate metamodel instances not only satisfying metamodel structural and OCL constraints but also meeting some specific criteria to generate more meaningful instances.

Thus, by analysing and understanding the papers we collected and described in this chapter, we can identify that three knowledge gaps which exist among those research domains, and thus can propose a new valuable research direction to the problem of automatic metamodel instance generation.

- There are no existing automated tools that naturally capture a metamodel's structure and also solve any OCL constraints defined for a metamodel. One may think that Alloy could be the best tool for generating metamodel instances. However, Alloy's specification is not designed for the purpose of capturing metamodel structures, it is designed for expressing relational specifications for more general constraint problems. Thus, it does not provide enough language features to support metamodels and instance selection criteria. Furthermore, SAT-based approaches are not particularly suitable for solving numeric constraints and these are very important to metamodel instance generation since one may require a specific number of nodes to be created in each instance.

- Graph grammars use a type graph representation to naturally capture a metamodel's structure, and SMT solvers are particularly good at solving not only boolean formulas but also numeric constraints. However, there is no research that combines a type graph representation with the SMT approach to naturally support metamodel structures and solve the OCL constraints.
- It would also be useful to construct such an extensible and fully automatic tool so that users can generate customised instances to meet specific criteria.

Therefore, a valuable research direction would be to combine the type graph representation used by graph grammars and SMT solvers to not only capture a metamodel's structure but also solve the OCL constraints. It would also be worthwhile to extend this direction to support coverage oriented instance generation.

2.4 Summary

In this chapter, we have reported on a systematic literature review on metamodel instance generation. The research domains that concern metamodel instance generation techniques were identified and papers from those research domains were discussed in detail with a view to answering our research questions. After analysing the existing research work, a potentially valuable direction has been pointed out. This is an approach that can combine both graph and SMT approaches to not only naturally support metamodel instance generation but also achieve both generic and specific coverage criteria for a metamodel.

Chapter 3

Background: Graphs, SAT and SMT

In this chapter, we review the essential concepts required for combining graph theory and SAT/SMT solvers, and applying them to the problem of automatic metamodel instance generation. These include those graph concepts that are closely related to metamodels, boolean satisfiability problems (SAT) and Satisfiability Modulo Theories (SMT) solvers. Graph concepts are well developed and are widely used to capture the structure of a metamodel [Ehrig et al., 2006], while SAT solvers and solvers with Satisfiability Modulo Theories built-in are particularly suited to solving constraint problems. This chapter also demonstrates the feasibility of using an SMT solver to solve complicated problems by showing an SMT-based Sudoku solver as an example.

In the following sections, we describe a set of graph concepts. These include *Typed Graph*, *Attributed Graph*, *Attributed Type Graph* and *Attribute Type Graphs with Inheritance*. In order to help the reader digest those concepts we also provide a list of the important notations used throughout this chapter in Table 3.1. The notations used in this section for graphs are the standard notations as described in [Ehrig et al., 2006]. Note that some of the graph concepts used here may refer to Category theory. We informally discuss these concepts in this chapter because we are only concerned with graph representations used for our instance generation. However, full details on Category theory can be found in the book

Notations	Description	Graph Type
V	The set of nodes	Graph (G)/Typed Graph (TG)
E	The set of edges	
s	Source function	
t	Target function	
V_G	The set of graph nodes	Attributed Graph (AG)
E_G	The set of graph edges	
E_{NA}	The set of node attributes	
E_{EA}	The set of edge attributes	
S_D	Attribute value sorts (types)	
D	Data signature algebra	
TG_{V_G}	The set of typed graph nodes	Attribute Type Graph (ATG)
TG_{V_D}	The set of typed data nodes	
TG_{E_G}	The set of typed graph edges	
$TG_{E_{NA}}$	The set of typed node attributes	
Z	Final data signature algebra	
$TG_{E_{EA}}$	The set of typed edge attributes	
I	Inheritance graph	Attributed Type Graph with Inheritance ($ATGI$)
A	A set of abstract nodes	

Table 3.1: A list of the important sets and functions used for introducing different types of graphs.

by Ehrig and Mahr [1985].

3.1 Metamodels and Graphs

Elements in a UML class diagram can be represented as a graph, where each class can be considered as a node, and the relationships between classes are edges linking one node to another. Since a metamodel can be defined using a UML class diagram, a metamodel can also be interpreted as a graph. We consider all metamodels used in this thesis as being presented as UML class diagrams and being formally represented as graphs.

3.1.1 Typed Graphs

A graph G is a 4-tuple $G = (V, E, s, t)$ contains a set of nodes or vertices V, and a set of edges E and two functions $s, t : E \to V$ which represent source and target functions respectively.

Graphs are related by graph morphisms that map the nodes and edges of a graph to those of another one, and preserving the source and target of each edge. Formally, given two graphs $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$ a graph morphism $f : G_1 \to G_2, f = (f_V, f_E)$ consists of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$. This means that a graph morphism f can map every node and edge in a graph G_1 to the nodes and edges in another graph G_2. Furthermore, f also preserves the source and target functions, that is $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$ [Ehrig et al., 2006] [Golas, 2011].

The valid instances of a metamodel are graphs that preserve type information about the classes in a metamodel. Thus, they are graphs typed over a metamodel (type graph). A graph G with a morphism *type*, where *type* is $G \to TG$ is called a typed graph. A typed graph TG is a 4-tuple where $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$, where V_{TG} and E_{TG} are called *typed* nodes and edges in a TG.

For example, given two graphs $G = (V_G, E_G, s_G, t_G)$ and $T = (V_T, E_T, s_T, t_T)$, then G will contain a morphism $type = (type_V, type_E)$ where

- $type_V : V_G \to V_T$
- $type_E : E_G \to E_T$

Figure 3.1 presents an example of graph G typed over graph T by its morphism *type*, defined as:

1. $type_V(a) = A$
2. $type_V(b) = B$
3. $type_E(e_1) = type_E(e_2) = E$

and morphism *type* preserves the source and target functions. For example, $type_V(s_G(e1)) = s_T(type_E(e1))$ and $type_V(t_G(e1)) = t_T(type_E(e1))$.

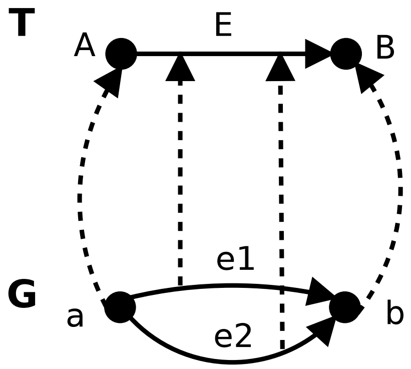


Figure 3.1: An example of typed graph. Dashed lines describe nodes and edges in the graph G and are typed over nodes and edges in the graph T.

3.1.2 Attributed Graphs and Attributed Type Graphs

Each class (node) in a metamodel may contain extra attributes for specifying more information about a class or an association (edge), thus a valid instance of that metamodel must also contain this information. To capture this information, a graph with attributes (attributed graph) is defined as follows [Ehrig et al., 2006]: An attributed graph is a tuple $AG = (G, D)$, with
$G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (s_j, t_j)_{j \in \{G,NA,EA\}})$ and $D = (S_D, OP_D)$ where

1. V_G denotes a set of *graph* nodes and V_D denotes a set of *data* nodes (vertices).

2. E_G, E_{NA} and E_{EA} denote a set of *graph edges*, *node attributes* and *edge attributes.*

3. $s_G : E_G \to V_G$, $t_G : E_G \to V_G$ are *source* and *target* functions mapping graph edges to nodes.

4. $s_{NA} : E_{NA} \to V_G$, $t_{NA} : E_{NA} \to V_D$ are *source* and *target* functions for the node-attribute edges.

5. $s_{EA} : E_{EA} \to E_G$, $t_{EA} : E_{EA} \to V_D$ are *source* and *target* functions for the edge-attribute edges.

6. $D = (S_D, OP_D)$ denotes a data signature algebra, where S_D denotes the implementation of a set of *attribute value sorts (types)* and OP_D denotes the implementation of a set of *operations* over S_D [Ehrig et al., 2006].

In the above definition, D uses concepts of signature and algebra. These concepts are fully derived from Category theory. For simplicity reasons, here we informally discuss these concepts from Category theory (as this section only concerns graph representations). The detailed and formal concepts can be found in [Ehrig and Mahr, 1985].

A signature gives the name for sorts and operations of an algebra. It is a syntactical description of an algebra. If we speak of a computer program, a signature can be considered as a formal description of the interface of that program. On the other hand, an algebra or a data signature algebra is an implementation of a particular signature. It implements the semantics of each sort and operation defined by a signature. Different types of mapping may exist between two algebras, and such mapping is referred to morphisms in Category theory [Ehrig and Mahr, 1985].

For example, the data signature algebra D may implement two signatures: $STRING$ and $INTEGER$. The $STRING$ signature may define a sort named *string*, and some operation names such as $concat : string\ string \rightarrow string$. S_D may implement *string* as a sequence of characters, and OP_D may implement the operation *concat* as follows:

$$\begin{aligned} concat: \quad & string \times string \rightarrow string \\ & (a, b) \mapsto ab \end{aligned}$$

This implies that operation *concat* joins two strings a, b together as a new string ab. The full descriptions for $STRING$ and INT can be found in [Ehrig et al., 2006].

Figure 3.2 presents an example of attributed graph. In this attributed graph the set of nodes and edges are formally given as follows:

1. $V_G = \{c1, m1, m2\}$

2. $V_D = \{2, class1, receive, send\}$

3. $E_G = \{c1m1, c1m2\}$

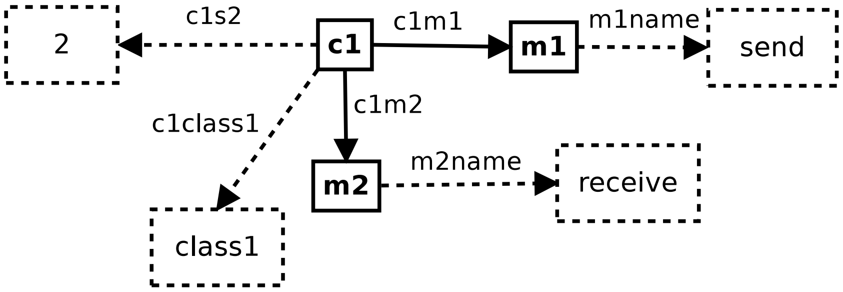


Figure 3.2: An example of attributed graph. The dashed nodes are the data nodes used in V_D, and the dashed lines are the edges linking graph nodes in V_G to data nodes in V_D

4. $E_{NA} = \{c1s2, m1name, m2name, c1class1\}$

5. $s_G(c1m1) = s_G(c1m2) = c1,\ t_G(c1m1) = m1, t_G(c1m2) = m2$

6. $s_{NA}(c1class1) = s_{NA}(c1s2) = c1,\ t_{NA}(c1s2) = 2, t_{NA}(c1class1) = class1$

7. $s_{NA}(m1name) = m1,\ t_{NA}(m1name) = send,\ s_{NA}(m2name) = m2,$ $t_{NA}(m2name) = receive$

Since there are no attributes for edges given in Figure 3.2, s_{EA} and t_{EA} are unused.

Since a metamodel is a type graph, and a type graph can also have attributes associated with its nodes and edges, an attributed type graph can also be used to depict a metamodel [Ehrig et al., 2006].

An attributed type graph is a tuple $ATG = (TG, Z)$, where $TG = (TG_{V_G}, TG_{V_D}, TG_{E_G}, TG_{E_{NA}}, TG_{E_{EA}}, (s_j, t_j)_{j \in \{TVG,TNA,TEA\}})$ is a type graph, where

1. TG_{V_G}: is the set of graph nodes in typed graph TG.

2. TG_{V_D}: is the set of data nodes in typed graph TG.

3. TG_{E_G}, $TG_{E_{NA}}$ and $TG_{E_{EA}}$ denote a set of *graph edges*, *node attributes* and *edge attributes* in the typed graph TG.

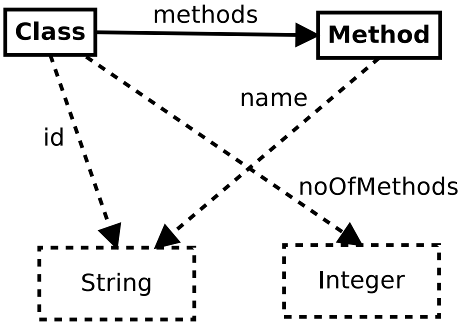


Figure 3.3: An attributed type graph for the attribute graph in Figure 3.2. The dashed boxes represent the different sorts. The dashed lines depict the links from the typed node to the sorts through different node attributes.

4. $s_{TVG} : TG_{E_G} \rightarrow TG_{V_G}$, $t_{TVG} : TG_{E_G} \rightarrow TG_{V_G}$ are *source* and *target* functions mapping graph edges to nodes.

5. $s_{TNA} : TG_{E_{NA}} \rightarrow TG_{V_G}$, $t_{TNA} : TG_{E_{NA}} \rightarrow TG_{V_D}$ are *source* and *target* functions for the node-attribute edges.

6. $s_{TEA} : TG_{E_{EA}} \rightarrow TG_{E_G}$, $t_{TEA} : TG_{E_{EA}} \rightarrow TG_{V_D}$ are *source* and *target* functions for the edge-attribute edges.

and Z is the final data signature algebra which means that for any arbitrary data signature algebra A, there is a unique mapping from A to Z [Ehrig et al., 2006] [Ehrig and Mahr, 1985]. The nodes and edges in an ATG represent the types that are used for the typing of an attributed graph.

As an example of an ATG, the graph in Figure 3.3 represents an attributed type graph for the attribute graph in Figure 3.2.

A typed attributed graph $(AG, type)$ contains an attributed graph AG along with a morphism $type : AG \rightarrow ATG$ [Ehrig et al., 2006], where $type = (type_{G,V_G}, type_{G,V_D}, type_{G,E_G}, type_{G,E_{NA}}, type_{G,E_{EA}}, type_D)$, where

- $type_{G,V_G} : V_G \rightarrow TG_{V_G}$, $type_{G,V_G}$ is a function that maps a graph node to a typed graph node in a typed attribute graph.

- $type_{G,V_D} : V_D \rightarrow TG_{V_D}$, $type_{G,V_D}$ is a function that maps a data node to a typed data node in a typed attribute graph.

- $type_{G,E_G} : E_G \rightarrow TG_{E_G}$, $type_{G,E_G}$ is a function that maps a graph edge to a typed graph edge in a typed attribute graph.
- $type_{G,E_{NA}} : E_{NA} \rightarrow TG_{E_{NA}}$, $type_{G,E_{NA}}$ is a function that maps a node attribute to a typed node attribute in a typed attribute graph.
- $type_{G,E_{EA}} : E_{EA} \rightarrow TG_{E_{EA}}$, $type_{G,E_{EA}}$ is a function that maps an edge attribute to a typed edge attribute in a typed attribute graph.
- $type_D : D \rightarrow Z$, $type_D$ is a mapping between two data signature algebras [1] [Ehrig et al., 2006; Ehrig and Mahr, 1985].

Following this definition, a morphism *type* is added to the attributed graph presented in Figure 3.2, and the following nodes and edges are derived from the attributed type graph depicted in Figure 3.3:

1. $type_{G,V_G}(c1) = Class,\ type_{G,V_G}(m1) = t_{G,V_G}(m2) = Method$
2. $type_{G,V_D}(2) = Integer,\ type_{G,V_D}(class1) = type_{G,V_D}(receive) =$ $type_{G,V_D}(send) = String$
3. $type_{G,E_G}(c1m1) = type_{G,E_G}(c1m2) = methods$
4. $type_{G,E_{NA}}(c1class1) = id,\ type_{G,E_{NA}}(cls2) = noOfMethods,$ $type_{G,E_{NA}}(m1name) = type_{G,E_{NA}}(m2name) = name$

3.1.3 Attributed Type Graph with Inheritance

As can be seen in Figure 3.3, a metamodel is depicted as an attributed type graph. However, a metamodel may contain an inheritance relationship between classes. Thus, in order to fully capture all structural elements of a metamodel, a graph that describes inheritance relationships is also needed.

An attributed type graph with inheritance is a triple $ATGI = (ATG, I, A)$ where

1. ATG is an attributed type graph.

[1] This mapping is called a homomorphism in Category theory.

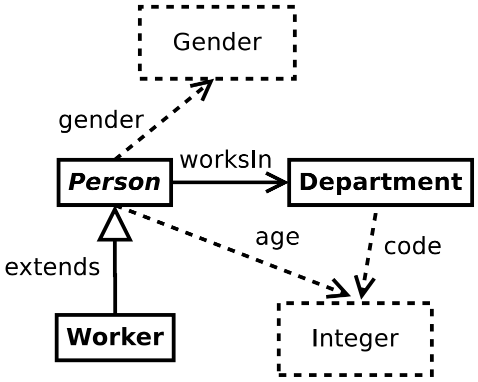


Figure 3.4: An example of an Attributed Type Graph with Inheritance

2. $I = (V_I, E_I, s_I, t_I)$ is an inheritance graph, where $V_I = TG_{V_G}$, $E_I = TG_{E_G}$, and $s_I, t_I : E_I \to V_I$.

3. A denotes a set of *abstract nodes*, where $A \subseteq V_I$.

For any node $x \in V_I$, $clan_I(x)$ is defined as $\{y \in V_I | \exists\ path\ y \overset{*}{\to} x\ in\ I\} \subseteq V_I$ with $x \in clan_I(x)$[Ehrig et al., 2006].

The attributed type graph with inheritance is similar as the attributed type graph except that the inheritance relationship between nodes are described by the graph I, and the set of abstract nodes are described by the set A. To illustrate these definitions using the $ATGI$ shown in Figure 3.4, the typed nodes and edges are listed below (we use the same notation as used in [Ehrig et al., 2006]):

$$\begin{aligned}
TG_{V_G} &= \{Person, Worker, Department\} \\
TG_{V_D} &= \{Integer, Gender\} \\
TG_{E_G} &= \{worksIn, extends\} \\
TG_{E_{NA}} &= \{gender, age, code, \} \\
TG_{E_{EA}} &= \emptyset
\end{aligned}$$

The inheritance relationship between typed nodes $Person$ and $Worker$ are described by the graph I:

1. $V_I = \{Person, Worker, Department\}$

2. $E_I = \{extends, worksIn\}$

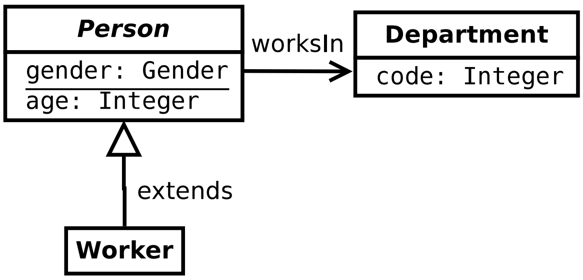


Figure 3.5: The Attributed Type Graph with Inheritance in Figure 3.4 in compact notation

3. $s_I(extends) = Worker$

4. $t_I(extends) = Person$

5. $s_I(worksIn) = Person$

6. $t_I(worksIn) = Department$

In Figure 3.4, the typed node $Person$ is an abstract node, thus $A = \{Person\}$. In addition, $clan_I(Person) = \{Person, Worker\}$, $clan_I(Worker) = \{Worker\}$, and $clan_I(Department) = \{Department\}$.

An $ATGI$ can also be represented in compact notation as shown in Figure 3.5. The elements of a metamodel in Figure 3.5 are captured by the definition of an $ATGI$. Thus, an $ATGI$ is a suitable concept for describing the complete structure of a metamodel.

Using the definition of an $ATGI$, a metamodel can be naturally captured, since the classes are typed nodes, relationships (associations and inheritance) are edges and fields in each class are attributes related to a particular typed node. The valid instances are attributed (typed) graphs typed over a metamodel. However, the structural and OCL constraints defined on a metamodel cannot be simply captured in this way. In fact, they define rules governing how a valid metamodel instance is formed, i.e. they address constraint problems. Among many techniques for solving constraint problems, SAT/SMT solvers are particularly suited for solving constraint and combinatorial problems, not only because they are well engineered but also because they come with fast solving speeds [Balint et al., 2013].

3.2 Boolean Satisfiability Problem

A boolean satisfiability (SAT) problem is a decision problem. Given a theory T that consists of a set of propositional boolean logical formulas $(F_1, F_2, ...F_n)$, the goal is to find an assignment that satisfies every single propositional formula. If such an assignment exists, a *model* of T is found. A computational procedure that can decide whether the set of boolean formulas is satisfied or not is called a *decision procedure.* However, the computation of such an assignment is not straightforward. The SAT problem has been proven to be the first known non-deterministic polynomial time complete (NP-Complete) problem [Cook, 1971].

Figure 3.6 shows an example of a SAT problem that is written in conjunctive normal form (CNF). In this formula, variables x_1, x_2, x_3 and x_4 are called *literals*, and each sub-formula with the form of $x_1 \vee \neg x_2 \vee \neg x_3$ is called a *clause.* The goal here is to determine whether there exists an assignment that satisfies every single clause.

In order to solve boolean satisfiability problems, many SAT solvers have been developed for determining the satisfiability of formulas similar to those shown in Figure 3.6 [Berre and Parrain, 2010; Een and Sörensson, 2005; Goldberg and Novikov, 2007; Mahajan et al., 2005; Moskewicz et al., 2001]. Many research domains use SAT-solvers as their back-end engines to solve difficult problems [Biere et al., 1999, 2003; Torlak, 2009]. A SAT solver takes in a large number of boolean formulas and determines their satisfiability. If the formulas are satisfied (sat), an assignment of each variable is returned. Otherwise, the SAT solver returns unsatisfied (unsat). Many SAT solvers are well engineered and can determine up to a million clauses within seconds using the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis and Putnam, 1960; Davis et al., 1962]. The DPLL algorithm is bounded by exponential time (EXP) which means that in the worst case it is very slow. However, in many real examples the algorithm is unlikely to run in EXP time. To solve the formula in Figure 3.6 is a very straightforward

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Figure 3.6: A propositional logic formula in conjunctive normal form.

task for SAT solvers. One of the possible satisfying assignments for this formula is $x_1 = true$, $x_2 = false$, $x_3 = true$ and $x_4 = true$.

SAT solvers can be used to solve a difficult problem by translating that problem into SAT. However, the difficulty here is that such a translation is not easy since a large number of propositional logic formulas are typically needed to express the problem. Encoding using plain boolean formulas lacks a power of expressiveness, e.g. it does not directly allow an integer encoding. For this reason, SMT is introduced to provide theories to express such problems without losing completeness and automation.

3.3 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem, like the boolean satisfiability (SAT) problem, is also a decision problem. An SMT problem can be expressed through a combination of many theories provided by SMT solvers. For example, SMT solvers can combine a quantified boolean theory, integer theory and bit-vector theory to specify a problem. Due to this support for a variety of theories, a problem can be relatively easy to translate to an SMT problem compared to translating it to a SAT problem.

Many SMT solvers are developed using the abstract DPLL framework. For each theory, a dedicated solving engine (decision procedure) is designed [Dutertre and de Moura, 2006; Nieuwenhuis et al., 2005, 2006], within the abstract DPLL framework. Solving an SMT problem expressed as a combination of different theories becomes the task of employing several particular solving engines to solve different sub-problems of that problem. In terms of their performance, SMT solvers can determine a large sized problem within seconds [Barrett and Tinelli, 2007; Cimatti et al., 2013; De Moura and Bjørner, 2008], and thus have been applied in many research domains such as model checking and program analysis [Armando et al., 2009; Cordeiro et al., 2009; Milicevic and Kugler, 2011; Tianhai Liu, 2012].

3.3.1 SMT-Lib version 2

SMT-Lib version 2.0 (SMT2) is the latest standard for SMT solvers, and it defines a common command language for specifying SMT problems [Barrett et al., 2010]. SMT2 uses a many-sorted first-order logic with equality as its underlying logic. For example, it supports the types *Int*, *Real* and *Bool* types.

3.3.1.1 Functions

The basic concept in SMT2 formulas is the total function. The command $(declare\text{–}fun\ f\ (a_1, a_2, ..., a_{n-1})\ (a_n))$ defines a function $f : a_1 \times a_2 \times ... \times a_{n-1} \to a_n$. A constant is simply a function f that takes no arguments. For example, $(declare\text{–}fun\ f\ ()\ T)$ defines a constant f.

3.3.1.2 Logic Context

A logic context l is specified using the command (*set–logic* l) before asserting any SMT2 formulas. The logic context indicates the combinations of background theories that are used for constructing SMT2 formulas. For example, logic QF_LIA provides the ability for users to write SMT2 formulas with a combination of boolean functions (QF) and linear integer arithmetic (LIA) functions.

3.3.1.3 Formulas

An SMT2 formula f can be asserted into the current logic context by using the command $(assert\ f)$. A simple SMT2 formula has one function application while a compound SMT2 formula involves multiple function applications. The current logic context determines special functions that can be used in the SMT2 formulas.

3.3.1.4 Models

SMT2 uses the command (*check–sat*) to check whether all SMT2 formulas asserted in the current logic context have a satisfying assignment. If the formulas are satisfied, the command (*get–model*) is used to retrieve a textual representation of an assignment. SMT2 also provides the command (*get–value*) to retrieve an assignment for the individual constants.

```
(set – logic QF_LIA)
(declare–fun x () Int)
(declare–fun y () Int)
(declare–fun z () Int)
(assert (>  (– z y)(+ x y))
(assert (=  z (∗ 2 x)))
(check–sat)
(get–model)
```

Figure 3.7: An SMT2 example that uses quantifier-free linear integer arithmetic logic to solve a inequality: $z - y > x + y$ and $z = 2x$.

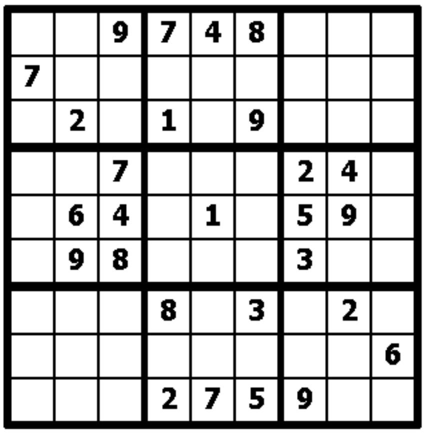
3.3.1.5 Solving An Integer Equation

Figure 3.7 shows an example of using quantifier-free linear integer arithmetic logic (QF_LIA) to solve the inequality $x + y > y + z$ with the extra constraint that $z = 2x$. If an SMT solver can find an assignment (model) to satisfy two SMT2 formulas, a model can be retrieved. In Figure 3.7, these constants (x, y and z) are declared, and two SMT2 formulas (using the *assert* command) are also specified within the current logic (QF_LIA).

As it can be seen, the arithmetic operators act like functions. For example, $+$ is a function that takes in 2 constants (x and y) as its arguments. After checking with the SMT solver using the command (*check–sat*), one possible model is retrieved using the command (*get–model*), that is when $x = 2$, $y = -1$ and $z = 1$, both formulas $z - y > x + y$ and $z = 2x$ are satisfied.

3.4 An SMT-based Sudoku Solver

SMT solvers are powerful solvers that are employed to solve complicated constraint problems such as a Sudoku puzzle. Software engineering and verification are full of problems like the Sudoku puzzle. We partially know the solution and try to figure out the rest, and it can be even worse when we are given nothing at all. One can utilise powerful SAT/SMT solvers as black-box solving/reasoning



		9	7	4	8			
7								
	2		1		9			
		7				2	4	
	6	4		1		5	9	
	9	8				3		
			8		3		2	
								6
			2	7	5	9		

Figure 3.8: An extreme-level Sudoku puzzle

engines to solve surprisingly difficult problems. However, one must know how to translate a problem to a SAT/SMT problem. Unfortunately, discovering such a translation is not always straightforward.

In this section, we demonstrate how to translate a Sudoku problem into an SMT problem in polynomial time. The Sudoku problem is NP-Complete [Yato and Seta, 2003]. It has rules stating that numbers 1 through 9 only appear once in each row, column and block (box with heavy solid lines). Each puzzle has a unique solution and some can be easily solved. Some are surprisingly hard to solve. For example, the Sudoku puzzle in Figure 3.8 may take hours to solve [M. Feenstra, 2013].

One can treat a Sudoku puzzle as a metamodel that is defined with a set of constraints expressing the rules for this game. Thus, each solution is a model or an instance for the Sudoku puzzle, and every instance has to obey the rules defined for this game. In order to use an SMT solver to solve a Sudoku puzzle, we analyse and group the game rules for Sudoku using the following constraints:

- The domain for the numbers in each cell is between 1 and 9.
- Numbers 1 through 9 can only appear once in each row.
- Numbers 1 through 9 can only appear once in each column.
- Numbers 1 through 9 can only appear once in each block.

- A solution has to contain the numbers that are already given by the puzzle.

The five constraints above state the general rules for playing Sudoku. In other words, every valid Sudoku solution must obey these constraints. Thus, these are *invariants* for every Sudoku puzzle solution. If one encodes these invariants into an SMT problem, an SMT solver will decide whether there is a solution to the puzzle or not. In other words, Sudoku problems can be translated into a decision problem which can be answered by an SMT solver.

To translate these invariants into an SMT problem, one can use linear integer arithmetic theory. For each cell in the puzzle, an SMT integer type constant is created. Thus, a total of 81 constants are needed. These invariants can be encoded into the SMT2 formulas in Figure 3.9. In the formulas in Figure 3.9, $|row|$ and $|column|$ denote the number of rows and columns in the Sudoku puzzle, $|block_{row}|$ and $|block_{column}|$ denote the number of rows and columns in each block, and $|number_{row}|$ and $|number_{column}|$ denote the number of rows and columns of a two-dimensional array that stores a list of initial numbers given by the puzzle.

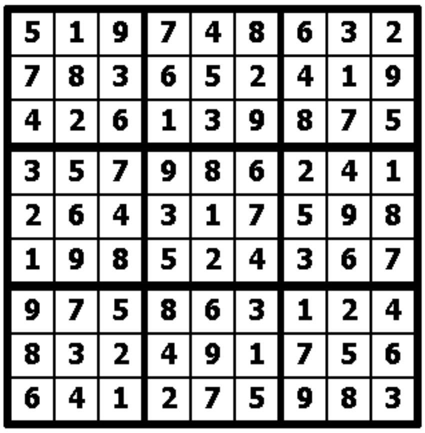
The first SMT2 formula requires 81 SMT integer constants representing each cell in the puzzle and limits their domain between 1 and 9. The second formula states that every cell in every row must be filled with a different number between 1 and 9 using inequality. In other words, the number in each cell in each row is *unique*. Similarly, the same encoding applies to each column and block. The last formula implies that the numbers that have already been given by the puzzle must be assigned to each corresponding cell. For example, the puzzle in Figure 3.8 has a total of 27 numbers given. Finally, these five formulas are conjoined and solved by an external SMT solver. After 133 milliseconds, the SMT solver successfully finds a solution (model) to the Sudoku puzzle in Figure 3.8. The solution is shown in Figure 3.10[2]. Each formula in Figure 3.9 can be implemented in an algorithm running in polynomial time and the detailed proofs can be found in Appendix 7.2.4. Therefore, we have transformed a Sudoku puzzle into an SMT2 problem in polynomial time. Solving the SMT2 formulas in Figure 3.9, we consequently solve the Sudoku puzzle.

[2] This SMT-based Sudoku solver is available at:
http://www.cs.nuim.ie/~haowu/ASMIG/Results/MM/Sudoku/

In the following formulas, the *row* and *column* denote the Sudoku puzzle's row and column respectively. Similarly, $block_{row}$ and $block_{column}$ denote the row and column of a block in the Sudoku puzzle. The initial values given by the puzzle are stored in the 2D-array *number*.

- $\bigwedge_{i=1}^{|row|} \bigwedge_{j=1}^{|column|} 1 \leq C_{i,j} \leq 9$
- $\bigwedge_{i=1}^{|row|} \bigwedge_{j=1}^{|column|-1} \bigwedge_{k=j+1}^{|column|} C_{i,j} \neq C_{i,k}$
- $\bigwedge_{i=1}^{|column|} \bigwedge_{j=1}^{|row|-1} \bigwedge_{k=j+1}^{|row|} C_{j,i} \neq C_{k,i}$
- For each block, we use the formula: $\bigwedge_{i=1}^{|block_{row}|} \bigwedge_{j=1}^{|block_{column}|} C_{i,j} \neq C_{k,l}$, where $i \neq k$ and $j \neq l$, k from 1 to $|block_{row}|$ and l from 1 to $|block_{column}|$. NOTE: when $i = 1$ and $j = 1$ means that the first cell in the block.
- $\bigwedge_{i=1}^{|number_{row}|} \bigwedge_{j=1}^{|number_{column}|} number_{i,j} \neq 0 \rightarrow C_{i,j} = number_{i,j}$, where *number* is a 2D-array, and $number_{i,j}$ denotes a number that is given at the *ith* row and *jth* column, and a blank cell is represented by 0.

Figure 3.9: SMT2 Formulas for the Sudoku puzzle



5	1	9	7	4	8	6	3	2
7	8	3	6	5	2	4	1	9
4	2	6	1	3	9	8	7	5
3	5	7	9	8	6	2	4	1
2	6	4	3	1	7	5	9	8
1	9	8	5	2	4	3	6	7
9	7	5	8	6	3	1	2	4
8	3	2	4	9	1	7	5	6
6	4	1	2	7	5	9	8	3

Figure 3.10: The solution found by SMT solver to the Sudoku puzzle in Figure 3.8

3.5 Summary

In this chapter, the graph concepts that are relevant to metamodeling techniques, as well as the boolean satisfiability problem (SAT) and Satisfiability Modulo Theories (SMT) have been reviewed. Essentially, metamodels are graphs and can be formalised using attributed type graphs with inheritance ($ATGI$), and SAT/SMT solvers are well engineered solvers that can solve a large number of formulas within seconds, which makes them popular in the software verification domain. The translation from a complicated problem to an SMT problem by using Sudoku as an example is demonstrated. The next chapter shows how the metamodel instance generation problem is translated to an SMT problem, and thus can be tackled by an SMT solver.

Chapter 4

Generating Metamodel Instances Satisfying Structural and OCL Constraints

The metamodeling approach provides a high-level abstraction for software engineers to model their systems. However, the central question that still needs to be addressed is: how one can automatically generate valid metamodel instances that satisfy structural and OCL constraints.

We represent metamodels as graphs with attributes and relationships (inheritance and associations), where fields define attributes that are related to nodes, and relationships constrain how the nodes are connected by edges. Generating metamodel instances is thus equivalent to generating nodes with relevant attributes and connecting edges between these nodes. Furthermore, a valid instance must also conform to the OCL constraints specified in the metamodel.

In this chapter, we introduce a novel approach that represents a metamodel as a bounded Attributed Type Graph with Inheritance (*ATGI*), derives a finite universe of all bounded attribute graphs typed over this bounded ATGI and translates this finite universe into SMT2 formulas which are solved by an external SMT2 solver. Each successful assignment for SMT2 formulas is interpreted as an instance of the original metamodel. This approach has been implemented into a new tool: *A Small Metamodel Instance Generator (ASMIG)*. We also evaluate

ASMIG against a list of different size metamodels and the results demonstrate that ASMIG is a very competitive tool in the area.

4.1 Bounded Attributed Type Graphs with Inheritance

In order to make sure the metamodel instance generation process terminates, we bound the search space and find all valid metamodel instances within that bound. To represent this, we develop a new graph concept: *a bounded attributed type graph with inheritance ($ATGI_b$).* This approach forms a finite universe of all bounded attribute graphs AG_u typed over a given $ATGI_b$, where this AG_u represents a superset of the instances that will be found.

Bounded Attributed Type Graph with Inheritance: A bounded $ATGI$ is a tuple $ATGI_b = (TG, Z, I, A, mult_s, mult_t, b)$, where

- (TG, Z) are the elements of an attributed type graph (ATG), and I, A are the elements of an ATGI (section 3.1.3).
- $mult_s$, $mult_t$: $TG_{E_G} \rightarrow \mathbb{Z}^+ \bigcup \{*\}$, where $*$ represents the multiplicity notation $*$ used in an association end.
  $mult_s$ and $mult_t$ are two functions which define the multiplicities at two association-ends.
- b is a function, $b : TG_{V_G} \rightarrow \mathbb{Z}^+$, defining a finite bound for each type node in TG with a constraint $\{\exists n \in \ TG_{V_G} \mid b(n) > 0\}$.

Since each ATGI contains a bounding function b, each instance of an $ATGI_b$ is bounded and finite. Thus, a bounded attributed graph typed over $ATGI_b$ is defined as $(AG_b, type_b)$ with the morphism $type_b : AG_b \rightarrow ATGI_b$, where $type_b = (type_{b,V_G}, type_{b,V_D}, type_{b,E_G}, type_{b,E_{NA}}, type_{b,E_{EA}}, type_{b,D})$ where

- $type_{b,V_G} : V_G \rightarrow TG_{V_G}$, maps V_G (the set of graph nodes in AG_b) to TG_{V_G} (the set of graph nodes in $ATGI_b$).
- $type_{b,V_D} : V_D \rightarrow TG_{V_D}$, maps V_D (the set of data nodes in AG_b) to TG_{V_D} (the set of data nodes defined in $ATGI_b$).

- $type_{b,E_G} : E_G \rightarrow TG_{E_G}$, maps E_G (the set of graph edges in AG_b) to TG_{E_G} (the set of graph edges in $ATGI_b$).
- $type_{b,E_{NA}} : E_{NA} \rightarrow TG_{E_{NA}}$, maps E_{NA} (the set of node attributes defined for V_G in AG_b) to $TG_{E_{NA}}$ (the set of nodes attributes in $ATGI_b$).
- $type_{b,E_{EA}} : E_{EA} \rightarrow TG_{EA}$, maps E_{EA} (the set of edge attributes defined for E_G in AG_b) to TG_{EA} (the set of edge attributes in $ATGI_b$).
- $type_{b,D} : D \rightarrow Z$, where D describes a data signature algebra in AG_b and Z describes the final data signature algebra in $ATGI_b$.

Each particular type node n in TG_{V_G} is assigned a bound by b, and this bound indicates the maximum number of instances of an *exact* instances of a type node n that may appear in each metamodel instance. An *exact* instance of a type node n means that we only consider type n, any child nodes that inherit from n are not considered as *exact* type n. A bound $b(n)$ for a type node n is either assigned manually by a user or automatically calculated according to the different multiplicities in the metamodel.

In order to automatically calculate an appropriate bound for each type node in TG_{V_G}, we have designed a Bound Calculator (this component is shown in Figure 1.4) that uses rules in Figure 4.1 to restrict the bounds based on different association-end multiplicities that have been specified. The Bound Calculator *automatically* translates these rules into SMT2 formulas, and invokes an SMT2 solver to find an appropriate assignment for the bound of each particular type node. Note that each rule in Figure 4.1 requires that the bound of A is greater or equal to 1 ($b(A) \geq 1$).

Rule 1 specifies that $b(B)$ (the bound of B) is greater or equal to 1. This is because it implies that every instance of A can only be associated with exactly one instance of B, but one instance of B can be associated with multiple instances of A. For a similar reason, we also require $b(B) \geq 1$ for the second association pattern since each instance of A is associated with at least one instance of B.

Similarly, we have also designed two rules (rule 3 and 4) for bidirectional association patterns. Rule 3 is for a *one–to–one* bidirectional association, we require the bounds for both A and B to be equal. This is because each instance of

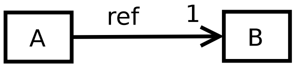
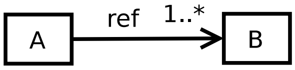
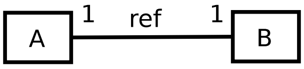
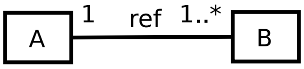
Association Pattern	Rule
	$b(B) \geq 1$ (1)
	$b(B) \geq 1$ (2)
	$b(A) = b(B)$ (3)
	$b(A) \leq b(B)$ (4)

Figure 4.1: Rules for bounding type nodes according to different association-end multiplicities. Here, we implicitly require $b(A) \geq 1$.

A is linked with exactly one instance of B, and each instance of B is also connected with exactly one instance of A. This implies that the number of instances of A and B are equal to each other. Rule 4 is for a *one–to–many* bidirectional association, this rule specifies that the bound of B is greater or equal to the bound of A. To see why it is, consider $b(B)$ is less than $b(A)$. This indicates that there are more instances of A than instances of B. Since each instance of A is linked with at least one instance of B, $b(A)$ is greater than $b(B)$ would suggest that at least one of the instances of A is not linked to any instance of B. Therefore, for this association pattern we require $b(B)$ to be greater or equal to $b(A)$.

For an abstract node, no explicit bound is assigned, but this can be calculated by summing the bounds of its concrete descendants. Thus, the bound n of a general or an abstract type of node n is calculated as the summation of each $b(m)$, where m is a child node of n. Ideally, $b(n)$ will be greater than zero, or else we consider that the type node n cannot be instantiated through its descendants.

With respect to the bound defined in $ATGI_b$, a *finite universe* of all bounded attributed graphs typed over $ATGI_b$ can be formed. Each instance of $ATGI_b$ can be derived from this universe.

Finite Universe: The finite universe of all bounded attributed graphs typed over $ATGI_b$ is defined as a pair $(AG_u, type_u)$, where (the notations used in the following text are summarised in Table 3.1)

1. AG_u is the finite universe of all bounded attributed graphs typed over $ATGI_b$.

2. $type_u : AG_u \to ATGI_b$, with
   $type_u = (type_{u,V_G}, type_{u,V_D}, type_{u,E_G}, type_{u,E_{NA}}, type_{u,E_{EA}}, type_{u,D})$, and where:

   - $type_{u,V_G} : V_G \to TG_{V_G}$, maps V_G (the set of graph nodes defined in the finite universe) to TG_{V_G} (the set of graph nodes in $ATGI_b$).
   - $type_{u,V_D} : V_D \to TG_{V_D}$, maps V_D (the set of data nodes in the finite universe) to TG_{V_D} (the set of data nodes defined in $ATGI_b$).
   - $type_{u,E_G} : E_G \to TG_{E_G}$, maps E_G (the set of graph edges in the finite universe) to TG_{E_G} (the set of graph edges in $ATGI_b$).
   - $type_{u,E_{NA}} : E_{NA} \to TG_{E_{NA}}$, maps E_{NA} (the set of node attributes defined for V_G in the finite universe) to $TG_{E_{NA}}$ (the set of node attributes in $ATGI_b$).
   - $type_{u,E_{EA}} : E_{EA} \to TG_{EA}$, maps E_{EA} (the set of edge attributes defined for E_G in the finite universe) to TG_{EA} (the set of edge attributes in $ATGI_b$).
   - $type_{u,D} : D \to Z$, maps a data signature algebra (D) in AG_u to the final data signature algebra (Z) in $ATGI_b$.

For example, Figure 4.2(a) shows an $ATGI_b$. The bound for each type node is depicted as a circled number in the top-right corner of each type node. Here, there is a bound of 2 for type node $Worker$, a bound of 1 for type node *Department* and no bound for abstract node $Person$. The multiplicity function $mult_t(worksIn)$ = $\{1\}$ and $mult_s(worksIn)$ are unused.

The finite universe of all bounded attributed graphs typed over the $ATGI_b$ depicted in Figure 4.2(a) is shown as follows:

For simplicity reason, we represent every edge in a form of $e = (a, b)$ in the rest of thesis, where a and b are source and target nodes of an edge e.

1. $V_G = \{w1, w2, d1\}$.
2. $V_D = \{age1, gender1, age2, gender2, code1\}$.
3. $E_{NA} = \{e_1 = (w1, age1), e_2 = (w1, gender1), e_3 = (w2, age2),$
   $e_4 = (w2, gender2), e_5 = (d1, code1)\}$.

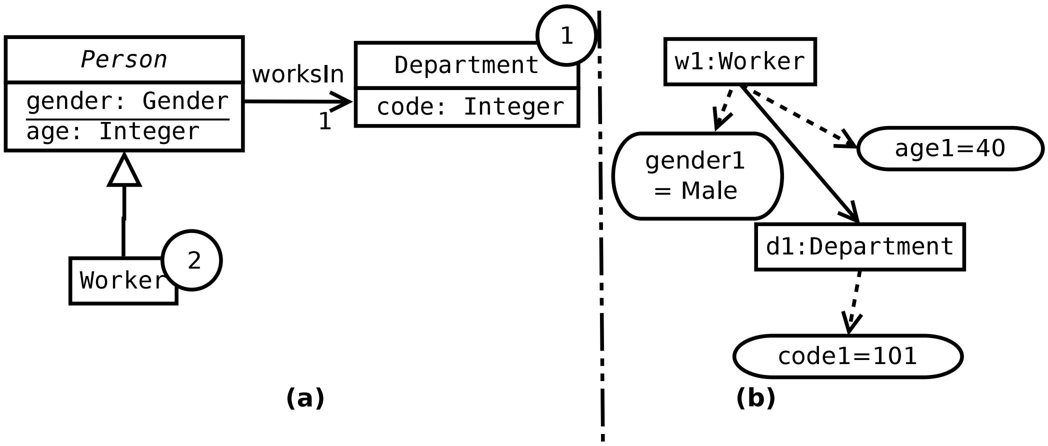


Figure 4.2: Examples of (a) a Bounded ATGI in compact notation. (b) an *instance* of a Bounded ATGI in explicit notation, where we have only selected one instance of $Worker$.

4. $E_G = \{e_6 = (w1, d1), e_7 = (w2, d1)\}$.

5. $type_{u,V_G}(w1) = type_{u,V_G}(w2) = Worker, type_{u,V_G}(d1) = Department$.

6. $type_{u,V_D}(age1) = type_{u,V_D}(age2) = type_{u,V_D}(code1) = Integer$.

7. $type_{u,V_D}(gender1) = type_{u,V_D}(gender2) = Gender$.

8. $b(type_{u,V_G}(w1)) = b(type_{u,V_G}(w2)) = 2, b(type_{u,V_G}(d1)) = 1$.

Figure 4.2(b) shows a sample instance derived from the universe of attributed graphs by selecting $\{w1\}$ from V_G, $\{age1, gender1, code1\}$ from V_D, $\{e_1,\ e_2,\ e_5\}$ from E_{NA} and $\{e_6\}$ from E_G, along with three assignments, of 40 to $age1$, 101 to $code1$, and $gender1$ to the literal $Male$.

4.2 Translating an AG_u to SMT2 Formulas

Since we represent a metamodel as an $ATGI_b$, metamodel instance generation becomes the process of instantiating an $ATGI_b$. Thus, our approach to generating metamodel instances is that nodes and edges defined in the finite universe (AG_u) of all bounded attributed graphs typed over $ATGI_b$ are translated into SMT2 quantifier free formulas, and the SMT2 solver is used to assign appropriate values for those nodes and edges. Then:

- each *successful* assignment by the SMT2 solver is interpreted as a bounded attributed graph typed over $ATGI_b$ (an instance of a metamodel).
- an *unsuccessful* assignment indicates that no graphs (instances) exist in the current bounds for each type node in $ATGI_b$. In this case the user can conclude that the metamodel is inconsistent in the current bounds. It is still, of course, possible to find an instance within larger bounds [Jackson and Damon, 1996].

4.2.1 Translating the Nodes and Edges

Figure 4.3 summarises the translation rules for graph nodes (V_G), data nodes (V_D), node attributes (E_{NA}) and edges (E_G) to SMT2 formulas. In this figure, rules 1-4 show the translation rules for translating nodes and edges, while rules 5-7 show the additional formulas for nodes and edges. In Figure 4.3,

- Every node in V_G is translated into a Boolean constant in SMT2, representing whether or not it will be present in the instance (rule 1).
- The nodes in V_D representing basic data types are translated into different types of constants according to their types (rule 2). The current approach supports three different basic data types which are Boolean, Integer and Enumeration type. A node in V_D which has an Integer type is directly translated to an Int constant in SMT2. Similarly, a Boolean type node is translated to a $Bool$ constant. A node whose type is an Enumeration type is translated to an SMT2 Int constant with an extra formula. This formula that limits the domain of an Int constant to between zero and the number of literals defined in an Enumeration data type, minus one.
- Each edge in E_G and E_{EA} is translated to a Boolean constant (rules 3 and 4), representing whether it will be present or not in the instance. For each edge e in E_{NA}, an additional formula is imposed to indicate that when an e is selected, both nodes $s_{NA}(e)$ and $t_{NA}(e)$ represented by e are also forced

to be selected (rule 5). Similarly, an additional formula is also needed for each edge in E_G (rule 6).

- For each data node d in V_D, an extra formula is also needed for specifying that when a graph node n in V_G is not selected, none of its data attributes need to be selected. The assignment for d is restricted to a single $fixed$ value v so that each time the graph node n is not selected, the value for the corresponding d is always a fixed default value. This formula is presented in rule 7 in Figure 4.3, with the $fixed$ value for d determined by its type.

4.2.2 Translating Graph Edges

A metamodel contains inheritance and association relationships between classes. These two relationships are additional constraints for a metamodel, translated as extra SMT2 formulas for the graph nodes, node attributes and graph edges contained in V_G, E_{NA}, and E_G.

We consider a generalisation relationship between two nodes $parent \in V_G$ and $child \in V_G$ with $clan_I(type_u(child)) \subset clan_I(type_u(parent))$. First, edges that represent all data nodes contained by $parent$ are encoded into a set of SMT2 constants called E_{child}. Second, E_{child} also encodes the set of edges that represent all data nodes contained by $type_u(child)$. For a $child$ that has two or more parents, data nodes contained by all parents are also encoded into E_{child}.

Associations in a metamodel are categorised into two kinds: unidirectional and bidirectional. These two kinds of associations are represented as edges in $ATGI_b$, and decorated with different multiplicities that impose a constraint on how two nodes are linked in a metamodel instance. To translate constraints for different multiplicities defined on an association ref, a set of SMT2 boolean constants E_{ref} is used to encode a subset of edges in AG_u (typed over the edge in $ATGI_b$) that represent an association is extracted from E_G (graph edges in AG_u). This E_{ref} encodes all the links between two different type nodes, we group them into a 2D-array, and then use logical connectives to specify the different multiplicities.

In this 2D-array:

- The rows and columns are captured by E_{row} and E_{col} respectively.

$$M_v : V_G \rightarrow SMT2\ constant$$
$$M_d : V_D \rightarrow SMT2\ constant$$

1. For each n_i in V_G, where $1 \leq i \leq |V_G|$ we generate
   (**declare–fun** $node_i$ () **Bool**),
   where $node_i$ is a unqiue name for each n_i in V_G,
   $|V_G|$ is decided by the bound allocated for a particular type node $m \in TG_{V_G}$ in $ATGI_b$ ($|V_G| = b(m)$).

2. For each d_i in V_D, where $1 \leq i \leq |V_D|$ we generate
   (**declare–fun** $data_i$ () **T**),
   where $data_i$ is a unique name for each d_i in $V_D, and\ T \in \{Bool, Int\}$.
   when $type_u(d_i) = Enum$ we use the following rule:
   (**declare–fun** $data_i$ () **Int**)
   (**assert** (**and** (>= $data_i$ 0) (<= $data_i$ $|Enum| - 1$)))
   Here $|Enum|$ denotes the number of literals in an enumeration type.

3. For each e_i in E_{NA}, where $1 \leq i \leq |E_{NA}|$ we generate
   (**declare–fun** $edge_i$ () **Bool**),
   where $edge_i$ is a unique name for each e_i in E_{NA}

4. For each e_i in E_G, where $1 \leq i \leq |E_G|$ we generate
   (**declare–fun** $edge_i$ () **Bool**),
   where $edge_i$ is a unique name for each e_i in E_G

5. For each e_i in E_{NA}, where $1 \leq i \leq |E_{NA}|$, the following extra formula is generated:
   (**assert** (=> $edge_i$ (**and** $M_v(s_{NA}(e_i))$ $M_v(t_{NA}(e_i))$))))

6. For each e_i in E_G, where $1 \leq i \leq |E_G|$, the following extra formula is generated:
   (**assert** (=> $edge_i$ (**and** $M_v(s_{NA}(e_i))$ $M_v(t_{NA}(e_i))$))))

7. For each e_i in E_{NA}, where $1 \leq i \leq |E_{NA}|$, the following extra formula is generated:
   (**assert** (=> (**not** $M_v(s_{NA}(e_i))$)) (= $M_d(t_{NA}(e_i))$ **v**)))
   where $s_{NA}(e_i) \in V_G, t_{NA}(e_i) \in V_D$
   $$\mathbf{v} = \begin{cases} -1 & \text{if } type_u(t_{NA}(e_i)) = Integer \\ false & \text{if } type_u(t_{NA}(e_i)) = Bool \\ 0 & \text{if } type_u(t_{NA}(e_i)) = Enum \end{cases}$$

Figure 4.3: Translation rules for translating graph nodes (V_G), data nodes (V_D), node attributes (E_{NA}) and graph edges (E_G) to SMT2 formulas.

- E_{row} encodes all links (edges) from one instance of A to one instance of B.
- E_{col} encodes all links (edges) from one instance of B to one instance of A.
- $e_{i,j}$ is an SMT2 boolean constant that encodes an edge from E_G, and placed at ith row and jth column.

4.2.2.1 Unidirectional Associations

Figure 4.4 summarises the translation rules for unidirectional associations. Here, only E_{row} is considered, this is because E_{row} encodes all links from one instance of A to one instance of B. Translation rule 1 in Figure 4.4 indicates that only one boolean constant can be selected from each row. Thus, this means each instance of A is associated with only one instance of B. Translation rule 2 is designed for an association ref with the multiplicity 0..1 defined at one end, this rule uses same formula used in rule 1 which make sure exactly one boolean constant from each row is selected. Since it is also possible that each instance of A is not associated with any instances of B, every boolean constant in each row of the array is also negated to indicate no links are selected. Finally, we join those two formulas by using disjunction to indicate that either no links get selected at all or exactly one link from each row is selected. Similarly, translation rule 3 uses disjunction to indicate that there are at least one of the links can be selected from each row. Thus, this captures the meaning of each instance of A is associated with at least one instance of B. For multiplicity 0..$*$, the constraint here indicates that either none of edges is selected or some of them are selected. Since each entry in 2D-array E_{ref} is an SMT2 boolean constant, each of them can be assigned with either value of $true$ or $false$. This precisely captures the meaning of either none or some of them. Therefore, we do not explicitly put this rule in Figure 4.4 but an explicit version of the formula can be seen in [Wu et al., 2013].

To understand how the translation rules work, we consider translation rule 1 as an example. Suppose we have already formed a 2D-array (as depicted in Figure 4.5) for the unidirectional association pattern. Let bounds for class A and B be 2 and 3 respectively. According to translation rule 4 described in Figure 4.3, in this 2D-array every entry is a boolean constant that encodes a possible link from an instance of A to an instance of B. Applying translation rule 1 through

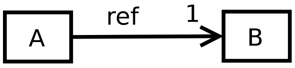
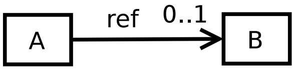
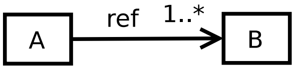
Association Pattern (Unidirectional)	Translation Rule
	$\bigwedge_{i=1}^{\lvert E_{row}\rvert} (\bigvee_{j=1}^{\lvert E_{col}\rvert} (\bigwedge_{k=1}^{\lvert E_{col}\rvert} k \neq j \rightarrow \neg e_{i,k}) \wedge e_{i,j}) \quad (1)$
	$(\bigwedge_{i=1}^{\lvert E_{row}\rvert} \bigvee_{j=1}^{\lvert E_{col}\rvert} \neg e_{i,j}) \bigvee$ $\bigwedge_{i=1}^{\lvert E_{row}\rvert} (\bigvee_{j=1}^{\lvert E_{col}\rvert} (\bigwedge_{k=1}^{\lvert E_{col}\rvert} k \neq j \rightarrow \neg e_{i,k}) \wedge e_{i,j}) \quad (2)$
	$\bigwedge_{i=1}^{\lvert E_{row}\rvert} \bigvee_{j=1}^{\lvert E_{col}\rvert} e_{i,j} \quad (3)$

Figure 4.4: Translation rules for graph edges typed over the edges in $ATGI_b$ that represent unidirectional associations.

$$ref = \begin{array}{c|ccc} & b_1 & b_2 & b_3 \\ \hline a_1 & e_{1,1} & e_{1,2} & e_{1,3} \\ a_2 & e_{2,1} & e_{2,2} & e_{2,3} \end{array}$$

Figure 4.5: A 2D-array represents an association ref, the bounds here for A and B are 2 and 3 respectively.

the 2D-array in Figure 4.5, we get the following two formulas (one for each row): f_1 and f_2.

$f_1 = (e_{1,1} \wedge \neg e_{1,2} \wedge \neg e_{1,3}) \vee (\neg e_{1,1} \wedge e_{1,2} \wedge \neg e_{1,3}) \vee (\neg e_{1,1} \wedge \neg e_{1,2} \wedge e_{1,3})$

$f_2 = (e_{2,1} \wedge \neg e_{2,2} \wedge \neg e_{2,3}) \vee (\neg e_{2,1} \wedge e_{2,2} \wedge \neg e_{2,3}) \vee (\neg e_{2,1} \wedge \neg e_{2,2} \wedge e_{2,3})$

Each f_i evaluates to true if only if exactly one of $e_{i,j}$ evaluates to true (i.e. is selected). Since the 2D-for ref has three columns, there are three possible choices in the disjunction. This pattern is also shown in Figure 4.6, where we can see that f_1 evaluates to true when exactly one of $e_{1,1}$, $e_{1,2}$ or $e_{1,3}$ evaluates to true. Since each $e_{i,j}$ encodes a possible link from an instance of A to an instance of B, this application of rule 1 to the 2D-array ref precisely captures the constraint that each instance A can only be associated with exactly one instance of B. Finally, we conjoin f_1 and f_2 as one complete formula, and input this to the SMT2 solver.

$e_{1,1}$	$e_{1,2}$	$e_{1,3}$	f_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figure 4.6: Truth table for formula f_1 where each 0 denotes false and 1 denotes true.

4.2.2.2 Bidirectional Association

Figure 4.7 summarises the translation rules for the most commonly used bidirectional associations. The rules for bidirectional associations are similar to unidirectional association. However, for a bidirectional association ref, the translation rules are applied in two dimensions (E_{row} and E_{col}) according to different multiplicities defined for both ends of the association ref. After applying the translation rules, each dimension results in one sub-formula, which is joined via conjunction.

Since a bidirectional link is symmetric, which encodes an edge of the form (a, b) is represented as two links, i.e. a link from a to b, and a link that goes back from b to a. Rule 1 describes a *one-to-one* bidirectional association. This is similar to rule 1 for unidirectional associations except that the translation rule is also applied through E_{col}, thus forcing the SMT2 solver to select exactly one $e_{i,j}$ from each dimension. Rule 2 is similar to rule 1 except that the first sub-formula in rule 2 ensures that no $e_{i,j}$s from a row are selected, and since all the links are bidirectional, the same sub-formula is applied through E_{col} to indicate either exactly one link from an instance of B is chosen or no links are chosen. Rule 3 conjoins a sub-formula from rule 1 with another sub-formula that allows at least one $e_{i,j}$ from each column to be selected. For rule 4, it is not necessary to constrain each row here since each instance of A can either connect to no instance of B or some instances of B, and every $e_{i,j}$ in the array is a boolean constant that could possibly to be selected or not by the SMT solver. Thus, for rule 4 we

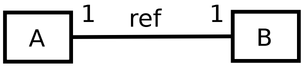
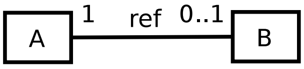
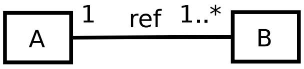
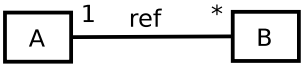
Association Pattern (Bidirectional)	Translation Rule
	$\bigwedge_{i=1}^{\lvert E_{row}\rvert} (\bigvee_{j=1}^{\lvert E_{col}\rvert} (\bigwedge_{k=1}^{\lvert E_{col}\rvert} k \neq j \rightarrow \neg e_{i,k}) \wedge e_{i,j}) \wedge$ $\bigwedge_{i=1}^{\lvert E_{col}\rvert} (\bigvee_{j=1}^{\lvert E_{row}\rvert} (\bigwedge_{k=1}^{\lvert E_{row}\rvert} k \neq j \rightarrow \neg e_{k,i}) \wedge e_{j,i}) \quad (1)$
	$(\bigwedge_{i=1}^{\lvert E_{row}\rvert} \bigvee_{j=1}^{\lvert E_{col}\rvert} \neg e_{i,j}) \bigvee \bigwedge_{i=1}^{\lvert E_{row}\rvert} (\bigvee_{j=1}^{\lvert E_{col}\rvert} (\bigwedge_{k=1}^{\lvert E_{col}\rvert} k \neq j \rightarrow \neg e_{i,k}) \wedge e_{i,j}) \wedge$ $(\bigwedge_{i=1}^{\lvert E_{col}\rvert} \bigvee_{j=1}^{\lvert E_{row}\rvert} \neg e_{j,i}) \bigvee \bigwedge_{i=1}^{\lvert E_{col}\rvert} (\bigvee_{j=1}^{\lvert E_{row}\rvert} (\bigwedge_{k=1}^{\lvert E_{row}\rvert} k \neq j \rightarrow \neg e_{k,i}) \wedge e_{j,i}) \quad (2)$
	$(\bigwedge_{i=1}^{\lvert E_{row}\rvert} \bigvee_{j=1}^{\lvert E_{col}\rvert} e_{i,j}) \wedge \bigwedge_{i=1}^{\lvert E_{col}\rvert} (\bigvee_{j=1}^{\lvert E_{row}\rvert} (\bigwedge_{k=1}^{\lvert E_{row}\rvert} k \neq j \rightarrow \neg e_{k,i}) \wedge e_{j,i}) \quad (3)$
	$(\bigwedge_{i=1}^{\lvert E_{col}\rvert} \bigvee_{j=1}^{\lvert E_{row}\rvert} \neg e_{j,i}) \vee \bigwedge_{i=1}^{\lvert E_{col}\rvert} (\bigvee_{j=1}^{\lvert E_{row}\rvert} (\bigwedge_{k=1}^{\lvert E_{row}\rvert} k \neq j \rightarrow \neg e_{k,i}) \wedge e_{j,i}) \quad (4)$

Figure 4.7: Translation rules for graph edges typed over the edges in $ATGI_b$ that represent bidirectional associations.

only need to constrain the column of the array to make sure each instance of B can only be connected to exactly one instance of A.

To understand how the translation rules work for bidirectional associations, we consider translation rule 1 as an example. We use the same 2D-array described in Figure 4.5 to represent all possible links between an instance of A and an instance of B. The bounds here for A and B are 2 and 3. By applying translation rule 1 through this 2D-array, we get the following formulas from each row and column.

$f_1 = (e_{1,1} \wedge \neg e_{1,2} \wedge \neg e_{1,3}) \vee (\neg e_{1,1} \wedge e_{1,2} \wedge \neg e_{1,3}) \vee (\neg e_{1,1} \wedge \neg e_{1,2} \wedge e_{1,3})$

$f_2 = (e_{2,1} \wedge \neg e_{2,2} \wedge \neg e_{2,3}) \vee (\neg e_{2,1} \wedge e_{2,2} \wedge \neg e_{2,3}) \vee (\neg e_{2,1} \wedge \neg e_{2,2} \wedge e_{2,3})$

$f_3 = (e_{1,1} \wedge \neg e_{2,1}) \vee (\neg e_{1,1} \wedge e_{2,1})$

$f_4 = (e_{1,2} \wedge \neg e_{2,2}) \vee (\neg e_{1,2} \wedge e_{2,2})$

$f_5 = (e_{1,3} \wedge \neg e_{2,3}) \vee (\neg e_{1,3} \wedge e_{2,3})$

From these formulas we can see that the pattern here is the same as applying translation rule 1 for the unidirectional association pattern except that this time the columns of the 2D array are also constrained. Since each column of the array encodes all possible links from an instance of B to an instance of A, by applying rule 1 through the column, we capture the meaning that each instance of B can only be associated with exactly one instance of A. This pattern is illustrated in f_3, $f4$ and f_5. For example, in f_3, if $e_{1,1}$ is selected, then this indicates that a link is established between a_1 and b_1. Thus, $e_{2,1}$ (a_2 is linked to b_1) can no longer be selected since each instance of B can only be associated with exactly one instance of A. Finally, we conjoin all these formulas and input to the SMT2 solver.

4.3 Translating OCL Invariants to SMT2 Formulas

Besides the constraints defined on multiplicities for associations, a metamodel can also have additional invariants expressed in OCL. Since everything in a metamodel is represented as an $ATGI_b$, we consider OCL invariants defined on a metamodel as additional formulas over the nodes and edges of the AG_u typed over $ATGI_b$. To deal with additional OCL invariants, we parse OCL invariants into an abstract syntax tree (AST), traverse the AST nodes to extract relevant nodes and edges from AG_u, and translate them into SMT2 formulas. We conjoin these formulas along with those produced for the metamodel structural constraints and transfer them to the SMT2 solver to find an assignment.

The OCL invariants are handled by the following translation rules:

1. OCL integer and logical expressions are directly translated to corresponding SMT2 functions.

2. Since a metamodel is represented as a bounded attributed type graph, quantifiers over an object type indicates a set of graph nodes (V_G) that are bounded by the type node (TG_{V_G}). The following translation rules define the translations from quantified OCL expressions to SMT2 formulas.

   - OCL expression $Obj.allInstances() \rightarrow exists(expr)$ is translated to

$$\bigvee_{i=1}^{b(Obj)} (expr_i)$$

where $expr_i$ is a specific SMT2 formula over an instance of Obj.

- OCL expression $Obj.allInstances() \rightarrow forAll(expr)$ is translated to
$$\bigwedge_{i=1}^{b(Obj)} (expr_i)$$
where $expr_i$ is a specific SMT2 formula over an instance of Obj.
- OCL expression $Obj.allInstances() \rightarrow one(expr)$ is translated to
$$\bigvee_{i=1}^{b(Obj)} ((\bigwedge_{j=1}^{b(Obj)} j \neq i \rightarrow \neg expr_j) \wedge expr_i),$$
where $expr_i$ and $expr_j$ are specific SMT2 formulas over an instance of Obj.

3. Users may use quantifiers in a nested way to indicate an operation over two different sets of instances. For this kind of nested quantifier expression, an extra translation step is performed by calculating the Cartesian product of two sets of graph nodes. The following translation rule shows the translation for a nested quantifier OCL expression.

- OCL expression $A.allInstances() \rightarrow forAll(B.allInstances()$ $\rightarrow exists(expr))$ is translated to
$$\bigwedge_{i=1}^{b(A)} \bigvee_{j=1}^{b(B)} (expr_i)$$
where $expr_i$ is a specific SMT2 formula over an instance of A and B.

4. A navigation used in an OCL expression is interpreted as a reference to a set of edges in the graph representation of a metamodel. The translation for a navigation r is performed by extracting relevant edges from E_G denoted as E_r, and using the following translation rule to translate them into SMT2 formulas.

- OCL expressions with navigation are translated to

$$\bigwedge_{i=1}^{|E_r|}(edge_i) \rightarrow (expr_i),$$

where $edge_i$ is an SMT2 constant that encodes an edge from E_G and each $expr_i$ is a specific SMT2 formula over an instance of $class$.

As can be seen, the translation rules above support a subset of OCL invariants and the supported OCL abstract syntax are also summarised in the Figure 4.8. To understand how the translation rules are applied to an invariant, Figure 4.9 shows an example of the full translation of an OCL invariant step by step. The expression $Manager.allInstances()$ indicates that all graph nodes having type $Manager$ are extracted from V_G, and the bound for $Manager$ here is 2. The OCL operators $>$,$<$ and *and* are translated to corresponding SMT2 functions, and the quantifier $exists$ is translated to *or* over the relevant instances.

4.4 A Graph Colouring Example

This section presents a metamodel of graph colouring as an example to demonstrate the approach. The metamodel itself is shown in Figure 4.10. This metamodel is constrained with two OCL invariants. The first one specifies that no node can have itself as its neighbour, while the second one indicates that a node and its neighbour cannot share the same colour. From the metamodel to $ATGI_b$, a finite universe AG_u typed over $ATGI_b$ to SMT2 formulas, each translation step is presented in Figure 4.11 and described as follows:

1. The metamodel itself is first translated into an $ATGI_b$ by defining $b(Node) = 5$ (as depicted in the right top corner of $Node$ in Figure 4.10), and a finite universe AG_u is formed from $ATGI_b$. We choose 5 for the bound of $Node$ because we consider a graph with 5 nodes has the appropriate size for presenting this example with regarding the page size here.

2. The typed node *Node* is translated to a Boolean constant in SMT2 with a defined bound of 5.

3. The data node *Colour* is an Enumeration type and thus gets translated into

$$
\begin{array}{lcl}
Expression & := & ExpAllInstances \\
 & & \mid ExpAttrOp \\
 & & \mid ExpConstBoolean \\
 & & \mid ExpConstEnum \\
 & & \mid ExpConstInteger \\
 & & \mid ExpNavigation \\
 & & \mid ExpQuery \\
 & & \mid OperationalExpression \\
ExpQuery & := & ExpExists \mid ExpForAll \mid ExpOne \\
OperationalExpression & := & [\mathit{expr1} : \mathit{Expression}]\ \mathit{ExpStdOp}\ [\mathit{expr2} : \mathit{Expression}] \\
ExpAttrOp & := & [\mathit{attr} : Type]\ [\mathit{expr} : \mathit{Expression}] \\
ExpAllInstances & := & [\mathit{source} : Type] \\
ExpStdOp & := & < \ \mid\ \leq\ \mid\ =\ \mid\ <>\ \mid >\ \mid\ \geq\ \mid \mathbf{or} \\
 & & \mid \mathbf{not} \mid \mathbf{and} \mid \mathbf{xor} \mid + \mid - \mid * \mid / \mid \rightarrow \\
ExpNavigation & := & [\mathit{src} : \mathit{ObjectType}]\ [\mathit{dst} : \mathit{ObjectType}]\ [\mathit{expr} : \mathit{Expression}] \\
ExpExists & := & [\mathit{var} : \mathit{Type}]^{*}\ [range : \mathit{Expression}]\ [query : \mathit{Expression}] \\
ExpForAll & := & [\mathit{var} : \mathit{Type}]^{*}\ [range : \mathit{Expression}]\ [query : \mathit{Expression}] \\
ExpOne & := & [\mathit{var} : \mathit{Type}]^{*}\ [range : \mathit{Expression}]\ [query : \mathit{Expression}] \\
ExpConstBoolean & := & [\mathit{const} : \mathit{BooleanType}] \\
ExpConstInteger & := & [\mathit{const} : IntegerType] \\
ExpConstEnum & := & [\mathit{const} : \mathit{EnumType}] \\
Type & := & \mathit{BasicType} \\
 & & \mid \mathit{EnumType} \\
 & & \mid \mathit{ObjectType} \\
\mathit{BasicType} & := & \mathit{BooleanType} \mid IntegerType \\
\mathit{BooleanType} & := & \mathbf{Boolean} \\
IntegerType & := & \mathbf{Integer} \\
\mathit{EnumType} & := & \mathbf{enum} \\
\mathit{ObjectType} & := & Class
\end{array}
$$

Figure 4.8: A summary of supported OCL abstract syntax. Note that we use the notation $[x : T]$ to denote that a variable x has a type of T and $[x : T]^{*}$ to denote a list of typed variables.

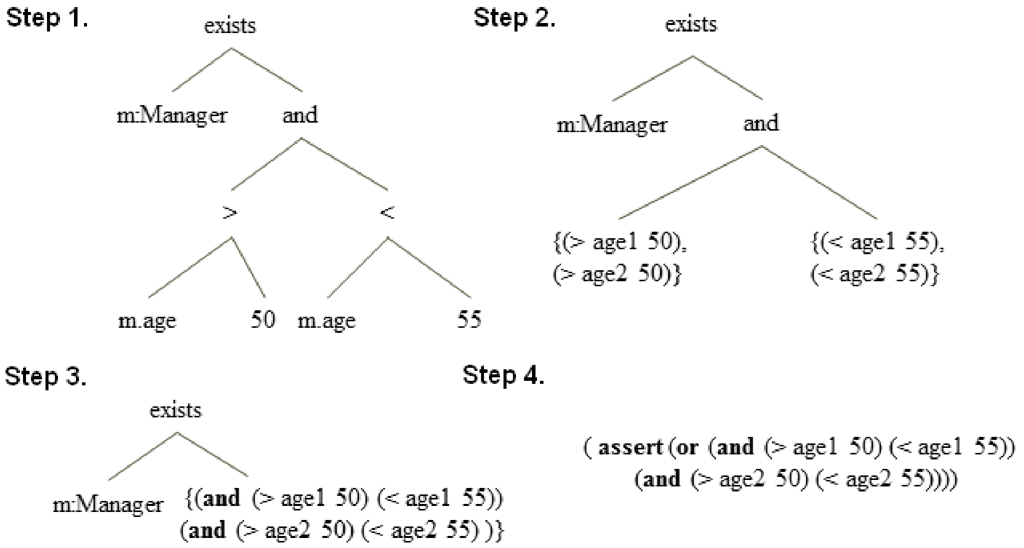


Figure 4.9: An example of translating an OCL constraint. Here, $b(Manager) = 2$ and the OCL constraint is: `Manager.allInstances()->exists(m|m.age>50 and m.age<55)`

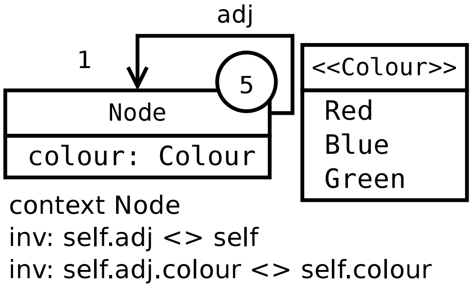


Figure 4.10: A Graph colouring metamodel

an Integer constant in SMT2 in the range $0 - 2$ inclusive (since there are only three colours here.).

4. Every edge in E_{adj} extracted from a subset of E_G is also translated to a Boolean constant in SMT2.

5. For the first invariant, the translation process eliminates all the edges e such that $s_G(e) \neq t_G(e)$. Thus, the graphs left are those that do not contain cycles of length 1.

6. The translation for the second invariant iterates the remaining edges, generating formulas to ensure that each $s_G(e).colour \neq t_G(e).colour$. Therefore, this step makes sure that every SMT2 constant that encodes edge's source and target object encoded are selected by SMT2 solver will not share the same value (literal).

7. Finally, the formulas from steps 2-4 are conjoined with formulas from step 5 and 6, and fed into an SMT2 solver. Each successful assignment found by the SMT2 solver is interpreted as an instance of the metamodel. Figure 4.12 shows one of the interpreted instances.

4.5 Evaluation

4.5.1 Implementation

To evaluate this approach, we have implemented it in a tool called *A Small Metamodel Instance Generator (ASMIG)*. ASMIG is purely written in Java and a fully automated tool. The main steps for ASMIG to generate metamodel instances is described as follows (can also be seen in Figure 1.4):

1. read in a metamodel in *Ecore* format [Budinsky et al., 2003],

2. represent it as a bounded attributed graph (AG_u) typed over $ATGI_b$,

3. translate AG_u and OCL invariants into logic formulas, rewrites logic formulas into SMT2 standard,

Step 1:
$V_G = \{n_1, n_2, n_3, n_4, n_5\}$
$V_D = \{c_1, c_2, c_3, c_4, c_5\}$
$E_{NA} = \{e_1 = (n_1, c_1), e_2 = (n_2, c_2), ..., e_5 = (n_5, c_5)\}$
$E_G = \{e_6 = (n_1, n_1), e_7 = (n_1, n_2), ..., e_{30} = (n_5, n_5)\}$
$type_u(n_1) = type_u(n_2) = ... = type_u(n_5) = Node$ and $Node \in TG_{V_G}$
$type_u(c_1) = type_u(c_2) = ... = type_u(c_5) = Colour$ and $Colour \in TG_{V_D}$
$type_u(e_1) = type_u(e_2) = ... = type_u(e_5) = colour$ and
$\forall i : 1 \leq i \leq 5\ type_u(s_G(e_i)) \in TG_{V_G}, type_u(t_G(e_i)) \in TG_{V_D}$
$type_u(e_6) = type_u(e_7) = ... = type_u(e_{30}) = adj$ and
$\forall i : 6 \leq i \leq 30\ type_u(s_G(e_i)) = type_u(t_G(e_i)) \in TG_{V_G}$

Step 2-4:
$V_G = (\textbf{declare–fun}\ n_1\ ()\ \textbf{Bool}), ..., (\textbf{declare–fun}\ n_5\ ()\ \textbf{Bool})$
$V_D = (\textbf{declare–fun}\ c_1\ ()\ \textbf{Int}), (\textbf{assert}(\textbf{and}(<=\ 0\ c_1)(<=\ c_1\ 2))), ...,$
$(\textbf{declare–fun}\ c_5\ ()\ \textbf{Int})\}, (\textbf{assert}(\textbf{and}(<=\ 0\ c_5)(<=\ c_5\ 2)))$
$E_{NA} = (\textbf{declare–fun}\ e_1\ ()\ \textbf{Bool}), ..., (\textbf{declare–fun}\ e_5\ ()\ \textbf{Bool})$
$E_G = (\textbf{declare–fun}\ e_6\ ()\ \textbf{Bool}), ..., (\textbf{declare–fun}\ e_{30}\ ()\ \textbf{Bool})$
$adj = (\textbf{assert}(\textbf{or}(\textbf{and}\ e_6\ (not\ e_7)...(not\ e_8)), ..., (\textbf{and}\ e_{26}\ (not\ e_{27})...(not\ e_{30}))))$
where each n_i, c_i and e_i is an SMT2 constant.

Step 5:
$(self.adj <> self) = (\textbf{assert}(\textbf{or}\ (\textbf{and}\ (not\ e_6),\ (not\ e_{12}),\ (not\ e_{18}),$
$(not\ e_{24}),\ (not\ e_{30})))),$
and where $e_6 = (n_1, n_1), e_{12} = (n_2, n_2), e_{18} = (n_3, n_3), e_{24} = (n_4, n_4), e_{30} = (n_5, n_5)$.

Step 6:
$(self.adj.colour <> self.colour) = (\textbf{assert}(=>\ \mathbf{e_7}\ (\textbf{not}(=\ \mathbf{c_1}\ \mathbf{c_2})))),$
$(\textbf{assert}(=>\ e_8\ (not\ (=\ c_1\ c_3)))), ...,$
$(\textbf{assert}(=>\ \mathbf{e_{11}}\ (\textbf{not}(=\ \mathbf{c_2}\ \mathbf{c_1})))), ..., (\textbf{assert}(=>\ e_{29}\ (not\ (=\ c_5\ c_4))))$

Figure 4.11: The illustrated translation steps for the metamodel in Figure 4.10. Note: In step 6, formulas written in red colour (e_7 and e_{11}) indicate that they are logically equivalent and they can be shared for efficiency.

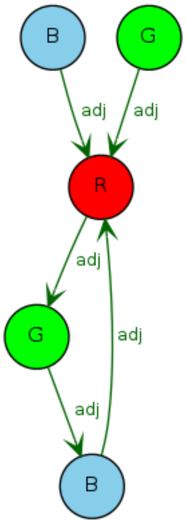


Figure 4.12: One of the instances of the graph colouring metamodel found by Z3. The letter in each node of this graph represents a colour (R:Red, G:Green, B:Blue), and an edge between two nodes means they are adjacent.

4. invoke an SMT2 solver (the default configuration uses the Z3 SMT2 solver) to find an assignment for SMT2 formulas,

5. interpret each successful assignment as a valid instance of the metamodel.

To speed up translation, all nodes and edges are stored in a hash table. To prevent generating repeated solutions for each enumeration, any previous successful assignments for a formula are negated and added as an extra SMT2 formula. ASMIG also supports partial models, but at present the partial model needs to be defined internally via relevant APIs. To visualise each instance that is generated, ASMIG generates a GraphViz representation for each successful assignment and caches the generated formula to speed up each enumeration [Ellson et al., 2001]. Furthermore, ASMIG can also generate instances that do not conform to the metamodel (negative test cases) by negating one or more of the SMT2 formula.

The interpretation from pure SMT2 formulas to the original metamodel instances is straightforward, as each SMT2 constant that encodes an edge or node is stored in a hash table. To fully interpret an assignment from SMT2 solver, we only need to iterate every entry in the hash table and reflect each assigned SMT2 constant back into the problem domain which is an instance of a metamodel. Thus, the time spent on interpretation is negligible compare to the time spent on translation.

4.5.2 Results

In order to evaluate the approach described in this chapter, we have collected a set of metamodels with different sizes from different sources, as shown in Table 4.1. These metamodels listed in Table 4.1 are well-known examples of the metamodeling approach such as state machine and royal & loyal [Warmer and Kleppe, 2003], [Ehrig et al., 2009]. In Table 4.1, "Classes", "Assocs" and "Attribs" indicates the number of (non-abstract) classes, associations and attributes translated for each metamodel. To evaluate the feasibility of this approach, the translation time for each metamodel is recorded. The average time spent on finding an instance is also calculated. This is based on the average time for the first 100 instances enumerated (except for the Finite State Machine 1.0 metamodel were there were only 16 instances enumerated in total with the bound of 1 for each class) as we consider it is a reasonable number for evaluating the average generation speed, comparing to other approaches [Cabot et al., 2008; Ehrig et al., 2009; Yatake and Aoki, 2012].

All instances for the metamodels are conducted on a machine with a 2.8GHz Intel Core2Quad CPU and 4GB of RAM. In the current version of ASMIG, Z3 is used as the back-end solving engine, so the average time spent on finding an instance depends on both the formulas generated and the Z3 solving time. All these metamodel instances are generated by using ASMIG with a default bound of 1 for each non-abstract class in 19 metamodels. In order to examine the translation for OCL invariants, bounds of 2 and 3 for classes in the *Company* metamodel are chosen.

[1] available at: http://www.emn.fr/z-info/atlanmod/index.php/Ecore

[2] an example extends the example metamodel in Figure 4.2, available at website

[3] from Eclipse Modeling Framework Royal and Loyal Example Project

[4] extracted from Eclipse Modeling Framework

[5] available at:http://www.jamopp.org/index.php/JaMoPP_Download

	Number of			Time in ms	
Metamodel	Classes	Assocs	Attribs	Translation	Avg Finding
Company [2]	7	6	6	476ms	38ms
C 1.0 [1]	34	4	0	388ms	54ms
C++ 1.0 [1]	16	4	5	372ms	26ms
Java [5]	233	104	1	666ms	441ms
Royal&Loyal [3]	15	41	2	403ms	36ms
Finite State Machine 1.0 [1]	6	7	0	368ms	26ms
Ecore [4]	22	40	0	439ms	42ms
UML2 Class Diagram [4]	40	26	46	442ms	41ms
Web App: Conceptual Model[1]	19	24	0	368ms	42ms
KM3 [1]	12	7	0	365ms	33ms
Business Process Model [1]	26	15	0	375ms	62ms
CPL1.0 [1]	32	16	0	384ms	94ms
DoDAF-SV5 [1]	31	54	1	391ms	99ms
GraphML [1]	11	13	2	392ms	37ms
Hierarchical State Machine 1.0 [1]	15	16	0	378ms	42ms
Maven(maven.xml) 0.3 [1]	58	32	0	403ms	74ms
MoDAF0.1 [1]	48	35	0	398ms	49ms
QualityofService [1]	24	26	0	376ms	51ms
DOT1.0 [1]	26	20	0	386ms	58ms
BibTexML1.2 [1]	28	4	0	379ms	39ms

Table 4.1: Details of 20 metamodels; 100 instances of these metamodels (except for the Finite State Machine) were generated by the ASMIG tool.

We have used metamodels collected from Table 4.1 as a benchmark to evaluate the speed of translation, average instance finding time and their practical applications. To analyse the relation between the translation time and metamodel size, we first define the size of metamodels is counted in terms of the number of classes. The reason we define this way is that we consider a metamodel with a large number of classes might have a good probability of having a complex structure. For example, multiple inheritance relationships, different types of attributes and associations or a large number of OCL constraints. This also makes it easier to consult Table 4.1 to do further analysis on translation time according to the specific numbers for a metamodel feature.

We plot a graph in Figure 4.13 to show the translation time against different size of metamodels. As Figure 4.13 suggested, the time that ASMIG spends is not always affected by the size of the metamodels. For example, ASMIG spends 398 milliseconds on translating the *MoDAF0.1* metamodel which has 48 classes, and spends 442 milliseconds on translating the *UML2 Class Diagram* metamodel which has 40 classes. By analysing both Figure 4.13 and Table 4.1, it is revealed that the translation time is in fact affected by three factors: the size of the metamodels, the type of associations in a metamodel and the number of OCL constraints defined. For example, a metamodel ($CPL1.0$) with large number of classes and less associations could spend less time on translation than a metamodel (*Ecore*) with relatively smaller number classes but with more associations. Regarding the type of associations used in a metamodel, a *one–to–one* bidirectional association produces more formulas than a *one–to–many* bidirectional association as it is more constrained. Having OCL invariants defined on a metamodel also takes longer time for ASMIG to translate. For example, the $Company$ metamodel has 7 classes, 6 OCL invariants, which require an additional $30ms$ for their translation. Thus, combining these three factors together decides the translation time ASMIG takes on a specific metamodel. All the instances generated from ASMIG cannot be presented due to the size, but these instances are available at our website [6].

[6] http://www.cs.nuim.ie/~haowu/ASMIG/Results/MM

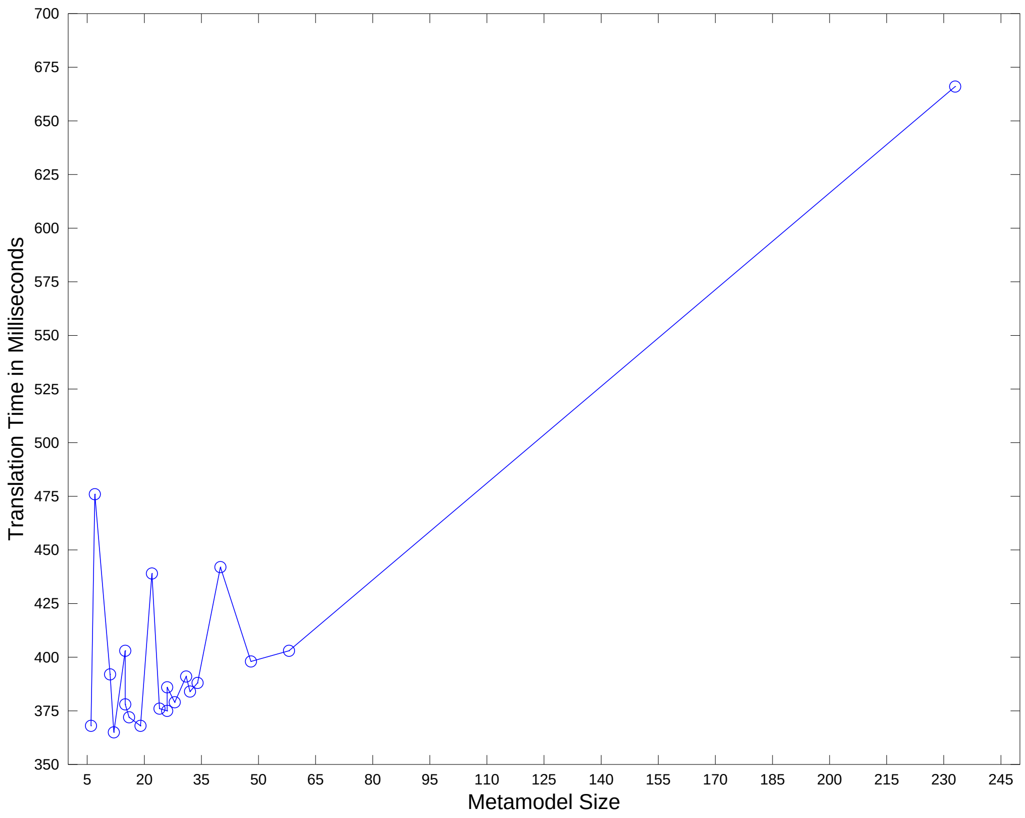


Figure 4.13: The time spent on translation by ASMIG with different size of metamodels. Each point in this graph represents the translation time that ASMIG takes on a metamodel from Table 4.1.

4.5.3 Comparison

We first provide the Table 4.2 to cover the comparison with those approaches that do not provide tool support. Second, we compare ASMIG with other main tools by using the same set of metamodels presented in Table 4.1.

In Table 4.2, the comparison is based on three factors: the size of metamodel, OCL range and automated tools. We choose these three factors because we consider they are the main concerns for a modeler to use a specific approach to generate instances from their metamodels.

The size of metamodels is an important aspect to measure the scalability of an approach. In Table 4.2, approaches like graph grammar and SMT bit-vector work on much smaller metamodels. These examples are specifically and manually designed [Ehrig et al., 2009] [Soeken et al., 2011b]. Compare with these approaches, our approach has been tested on a wide range of metamodels collected from different sources as can be seen from Table 4.1. Constraint programming and Alloy-based approaches work on relatively large metamodels and they also provide tool support, thus we compare these two approaches with our tool ASMIG, and discuss the results later in this section.

For the factor of OCL, all approaches support a certain amount of OCL. Among them, Alloy-based approaches support quite a wide range of OCL. Alloy is a fully automatic model finder that uses first-order relational algebra as its specification language and a SAT solver as its back-end solving engine, this research originates from the Jackson's work [Jackson, 2002] [Jackson, 1998] [Jackson and Damon, 1996]. With Alloy's specification language, quite a number of OCL constraints can be directly mapped into that language. However, this language lacks support for solving numeric constraints. This is because solving numeric constraints is beyond the capability of SAT solvers. Numeric constraints appear quite frequently in OCL. For example, in a company metamodel, employers may require the calculation of the payment for employees based on their different roles. On the other hand, SMT-based approaches such as SMT-bit vector and ASMIG are much stronger at handling numeric constraints compared to SAT-based approaches. This is because SMT solvers have particularly well engineered decision procedures to solve numeric constraints. SMT-bit vector provides a list of map-

Approaches	**Metamodel Size**	**OCL Range**	**Tool Automation**
Our Approach	Large	Medium	Yes
Graph Grammar	Small	Medium	No
SMT bit-vector	Small	Medium-high	No
Constraint Programming	Medium	Medium	Yes
Alloy-based Approach	Medium	High	Yes

Table 4.2: A comparison with other approaches.

ping from OCL collection data types to SMT bit vector theories. However, users have to manually write SMT formulas in order to perform such mapping, and this becomes impossible when users are dealing with a large number of constraints and are not familiar with any SMT theories.

To be able to effectively compare ASMIG with other tools, we use the same set of metamodels presented in Table 4.1 as the benchmark. We select three tools: UML2CSP, EMF2CSP and Echo (UML2CSP and EMF2CSP are constraint programming approach based, and Echo is based on Alloy). We choose these tools not only because they are available to access but also they represent the latest techniques for generating metamodel instances, and they also share the same input format (ecore) as ASMIG does. For each tool, we test them against our benchmark, generate one instance for each metamodel and record its translation time. The translation time for each tool spent on each metamodel is shown in Table 4.3. In Table 4.3, we use an F to denote that a specific tool fails to translate a metamodel. We also plot a graph in Figure 4.14 to show the translation time difference among all four tools. For a fair comparison, Figure 4.14 only shows these metamodels that are successfully translated by all tools.

For constraint programming approach, we download their latest tools: UML2CSP [7] and EMF2CSP [8], and run them against our benchmark [Cabot et al., 2008][González Pérez et al., 2012]. Both tools failed to translate a total of 8 metamodels into constraint search problems without giving specific error messages. Thus, we do not know the reason why those 8 metamodels cannot be translated. Regarding the translation time for supported metamodels, both tools are much slower than ASMIG, as can be seen from Figure 4.14. The main reason is that both EMF2CSP and UML2CSP require ECL^{i}PS^e compilation for the translated code from a metamodel and then solve the constraints. However, ASMIG's translation process does not require any third-party tool to do such compilation, because all metamodels are naturally supported by our Bounded Attributed Type Graphs with Inheritance.

We choose Echo [9] for Alloy-based approaches because it uses the latest engine of Alloy (kodkod) to generate metamodel instances and reads in a metamodel in ecore format [Macedo and Cunha, 2013]. As can be seen from Table 4.3, there are a total of 4 metamodels that cannot be supported by Echo. Two (*Java* and *UML2 Class Diagram*) cannot be supported because of multiple inheritance relationships. Echo failed to give additional error details for the other two metamodels (*Ecore* and *Royal &Loyal*). The reason Echo does not support multiple inheritance relationships is because Alloy's specification language only allows users to define single inheritance relationship from a particular class. This puts a great limitation on generating metamodel instances since multiple inheritance relationship is a common type of inheritance used in metamodeling approach. In Figure 4.14, the translation time for Echo and ASMIG on small and medium size metamodels is quite close. This indicates that translating a metamodel to SAT and SMT is more efficient than translating it to constraint search problem.

In summary, the results and comparison show that ASMIG is very competitive against other well-automated tools with regard to the support of metamodel size, OCL range and speed of translation. Note that ASMIG is purely designed from scratch and not built upon other existing tools such as Alloy. Therefore, ASMIG takes full control of the metamodel instances generation without tuning other tools. This means ASMIG is not limited by the capabilities of other tools such as lacking multiple inheritance support for Echo.

[7] http://qres.uoc.edu/UMLtoCSP

[8] http://code.google.com/a/eclipselabs.org/p/emftocsp/

[9] http://haslab.github.io/echo/

Metamodels	**Translation time for different tools in ms**			
	ASMIG	UML2CSP	EMF2CSP	Echo
Company	476	*F*	*F*	520
C 1.0	388	*F*	*F*	354
C++ 1.0	372	*F*	*F*	370
Java	666	*F*	*F*	*F*
Royal&Loyal	403	775	676	*F*
Finite State Machine 1.0	368	650	612	377
Ecore	439	*F*	793	*F*
UML2 Class Diagram	442	*F*	*F*	*F*
Web App: Conceptual Model	368	740	743	369
KM3	365	502	485	357
Business Process Model	375	702	707	377
CPL1.0	384	*F*	*F*	386
DoDAF-SV5	391	697	716	440
GraphML	392	498	448	408
Hierarchical State Machine 1.0	378	530	458	370
Maven(maven.xml) 0.3	403	793	884	429
MoDAF0.1	398	753	783	400
QualityofService	376	458	481	398
DOT1.0	386	565	569	380
BibTexML1.2	379	*F*	*F*	364

Table 4.3: A comparison with other well-automated tools: UML2CSP, EMF2CSP and Echo. In this table, the same set of metamodels from Table 4.1 are used, and an *F* denotes an individual tool that fails to translate a metamodel.

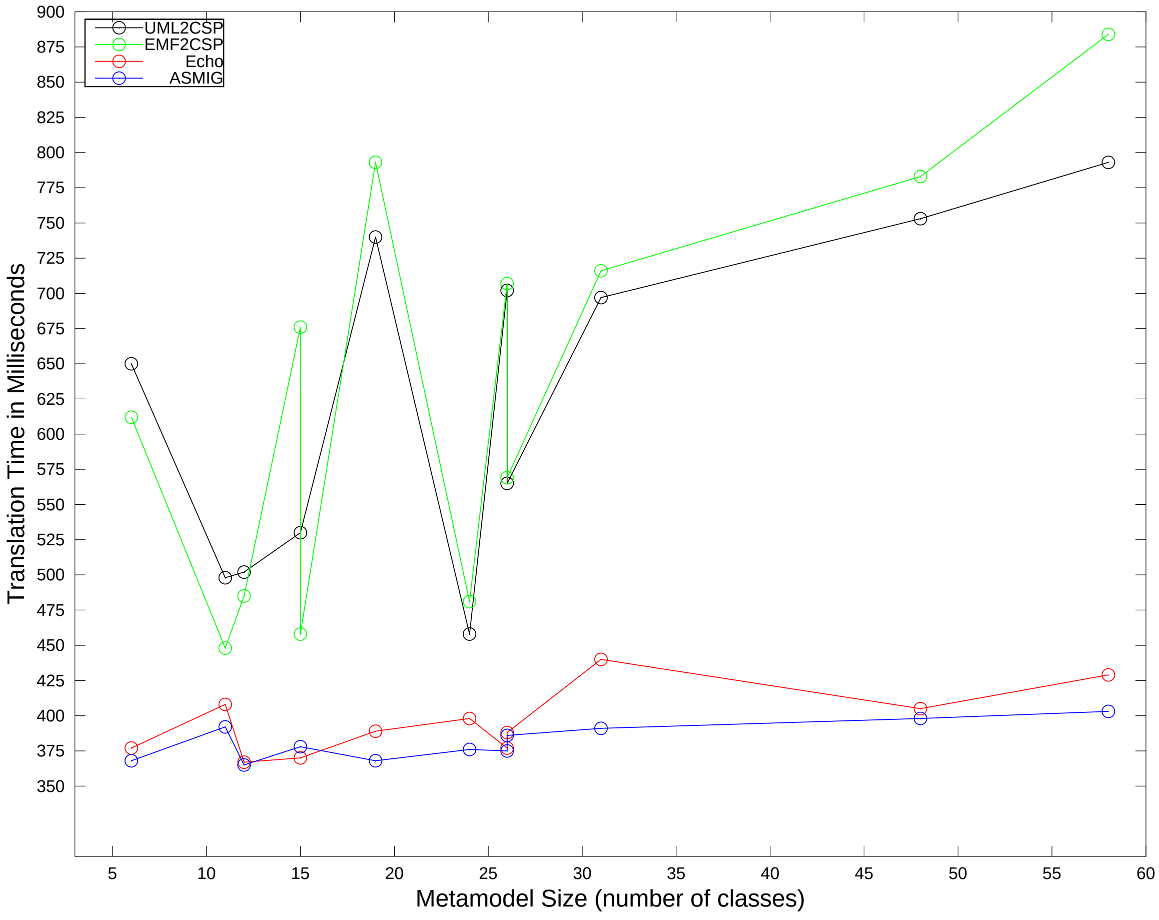


Figure 4.14: A comparison with UML2CSP, EMF2CSP and Echo. Each point in this graph represents the translation time that individual tool takes on a specific metamodel in Table 4.3. Note that only those metamodels that are successfully translated by the tools listed in Table 4.3 are included.

4.6 Summary

This chapter presents a novel approach that can generate metamdoel instances by developing a bounded attributed type graph with inheritance ($ATGI_b$), using it as an intermediate representation and translating the universe (AG_u) derived from $ATGI_b$ to SMT2 formulas. These SMT2 formulas are solved by an SMT2 solver, and each successful assignment is interpreted back into an instance of the original metamodel. This process has already been successfully automated into a tool *A Small Metamodel Instance Generator (ASMIG).* ASMIG reads in a metamodel in ecore format, along with a subset of OCL constraints, generates a set of SMT2 formulas and uses an external SMT2 solver to find an assignment for the SMT2 formulas. The results evaluated for ASMIG show the feasibility of this approach. The comparison with other main approaches show ASMIG is a promising tool that is capable of handling large size metamodels, a good amount of OCL constraints and decent translation speed. However, this approach does not consider any testing criteria for a metamodel, since a user may desire more meaningful instances to test their metamodels. This leads to an extension of the approach described in this chapter, directed towards producing more meaningful metamodel instances.

Chapter 5

Generating Metamodel Instances Satisfying Partition-Based Coverage Criteria

An effective technique for generating instances of a metamodel should quickly and automatically generate instances satisfying the metamodel's structural and OCL constraints. Ideally it should also produce quantitatively meaningful instances with respect to certain criteria, that is, instances which meet specified generic coverage criteria that help the modelers test or verify a metamodel at a general level.

This chapter describes a technique that extends the approach described in the previous chapter to produce such meaningful instances. This technique can generate meaningful instances based on partition-based criteria which must be provided by users [1]. A set of translation rules have been developed to translate these criteria to SMT2 formulas which are then solved by an SMT2 solver. Each successful assignment is then interpreted as a metamodel instance that provably satisfies a coverage criterion. This technique has been automated into our ASMIG tool, and the results of our evaluation of ASMIG demonstrate its feasibility.

[1] Note: the partition-based coverage criteria used in this chapter must be manually provided by the users.

5.1 Partition-based Coverage Criteria for Metamodels

Equivalence partitioning is a standard approach to software testing that divides data into equivalence classes, and seeks to ensure that test cases cover these classes. This approach has been extended to UML class diagrams [Andrews et al., 2003].

The coverage criteria defined for a UML class diagram can also be applied to a metamodel, since a metamodel can be represented as a UML class diagram. In particular, we are interested in the coverage criteria based on equivalence partitioning defined by Andrews et al. These partition-based criteria that they present are:

- Association-End Multiplicity coverage (AEM) which measures association relationships defined between classes.
- Class attribute coverage (CA) which measures the set of representative attribute value combinations in each instance of class.

AEM and CA are partition-based testing criteria which means that testing results depend on the choice of a single value from each partition. All other values are expected to yield the same results.

A metamodel typically describes the structural elements of classes using attributes from some given types. For example, in Figure 5.1, the attribute *methodCount* records the total number of methods contained in a class. The CA coverage described for an attribute with an integer type could be achieved using three partitions (< 0, $= 0$ and > 0) [Andrews et al., 2003]. Thus, to satisfy the CA criterion for this metamodel a tester might wish to generate instances with $methodCount < 0$, $methodCount = 0$ and $methodCount > 0$. One might consider that *methodCount* should be always greater or equal to 0. In this case, a specific value v ($v > 0$) can be chosen to form three partitions: $methodCount < v$, $methodCount = v$ and $methodCount > v$, as this is shown in Section 5.2.1.1. The implicit assumption is that any single value from one of the three partitions is sufficient to test all other values from that partition.

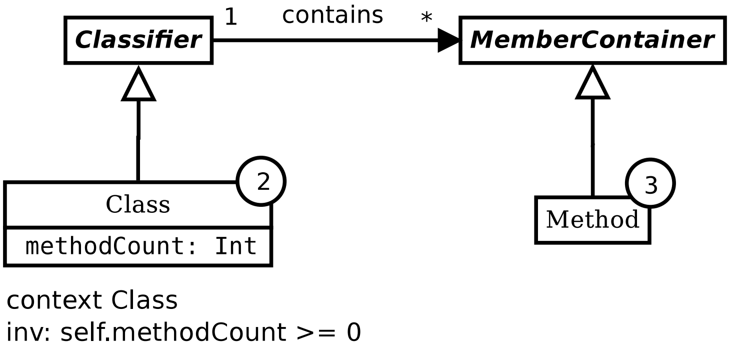


Figure 5.1: A subset of a programming language metamodel, depicted using bounded ATGI notation. The number in each circle represents the bound on the number of instances for a particular class (2 and 3 for *Class* and *Method* respectively).

The AEM coverage criterion for an association defines how instances can be divided into partitions according to the association-end multiplicities defined for two association-ends. Specifically, the number of partitions for AEM coverage is calculated by taking the Cartesian product of the multiplicities defined for two associations ends. For example, the binary association *contains* in Figure 5.1 has multiplicities 1 and $*$ defined at each end of the association, specifying that a *Class* can have multiple (including zero) *Method*s, but a *Method* belongs to exactly one *Class*. Thus, the Cartesian product of $\{1\}$ and $\{*\}$ is $\{(1, 0), (1, MAX)\}$. Here MAX defines the maximum number of *Methods* that can be contained in a *Class*, where this value can either be user-defined or default to the maximum integer value. Thus, to satisfy the AEM criterion for the metamodel in Figure 5.1, two instances are needed: one instance where a *Class* has no *Methods* and another instance where a *Class* has the maximum number of *Methods*.

5.2 Using Partition Switches and Criterial Formulas for Partition-based Instance Generation

To extend the approach as described in Chapter 4, we introduce a partition set P which contains all the features to be partitioned from the finite universe (AG_u). We use unquantified linear integer arithmetic (QF_LIA), described in Section 3.3.1.5), as a back-end theory for all generated SMT2 formulas, and limit ourselves to metamodels with OCL constraints as described in 4.3.

The general form of a translation rule described in this chapter is captured by the following template:

$$\bigwedge_{i=1}^{|P|} \bigvee_{j=0}^{|P_i|-1} (T_i = V_j) \wedge F_i$$

where

- P is the set of features to be partitioned in the graph, and $|P|$ is the cardinality of this set.
- P_i is an element in the set P, either from V_D or E_G.
- $|P_i|$ denotes the total number of partitions for P_i.
- T_i is a *partition switch* (explained below) for an element P_i in P, and it ranges from 1 to $|P_i|$.
- V_j is an integer that ranges from 0 to $|P_i| - 1$.
- F_i is a $criterial\ formula$ (explained below) that is used in conjunction with the partition switch.

All translation rules described in this chapter consist of two elements: a partition switch and a criterial formula (marked with blue colour in each translation rule). The template for generating a partition switch for each partition created for P_i is shown in Figure 5.2.

For every partition created for P_i, where $P_i \in V_D$ or E_G, generates:
$(\textbf{declare} - \textbf{fun}\ T_i()\textbf{Int})$
$(\textbf{assert}\ (1 \leq T_i \leq |P_i|))$
$(\textbf{assert}\ ((T_i = V_j)\ \wedge\ F_i))$
$(\textbf{assert}\ (0 \leq V_j \leq |P_i| - 1))$
where T_i is a partition switch for each partition created for P_i,
V_j is an integer and F_i is a criterial formula.

Figure 5.2: The SMT2 template for generating a partition switch used in each translation rule.

A *partition switch* P_i determines when a particular partition is to be switched on or off based on the integer value V_j. This implies that the value for V_j chosen by an SMT2 solver determines which particular partition P_i is selected.

A *criterial formula* F_i determines how elements to be selected from the universe correspond to the particular partition P_i. In other words, F_i puts extra constraints on the entire formula to specify how nodes and edges are appeared in an instance to satisfy a partition. The conjunction between a partition switch and a criterial formula guarantees that the criterial formula has to be applied on a particular partition (if that partition is to be covered).

For different partition-based criteria, ensuring that the instances generated by the SMT2 solver achieve those criteria, depends on criterial formulas in each translation rule. These criterial formulas constrain nodes and edges in a graph, and are described in the sections 5.2.1 and 5.2.2.

5.2.1 Class Attribute Partitioning

Achieving CA coverage requires that a test suite covers every partition created for each attribute. Since a metamodel is represented using a bounded attributed typed graph with inheritance ($ATGI_b$), the set of attributes in each class corresponds to the set of *data nodes* in V_D in the universe. Thus, partitioning an attribute in a class is the same as partitioning a data node in a graph. The criterial formulas determine what value is to be assigned for a data node.

The translation rule for a data node presented in Figure 5.3 contains criterial formulas on two different types of attributes (data nodes): integer and boolean.

$$M_d : V_D \rightarrow SMT2\ constant$$

- when $type_u(d) = \textbf{Int}$ generate:
  $(\textbf{assert}\ ((T_i = 0) \wedge (M_d(d) < p))\ \vee\ ((T_i = 1) \wedge (M_d(d) = p))$
  $\vee\ ((T_i = 2) \wedge (M_d(d) > p)))$
- when $type_u(d) = \textbf{Bool}$ generate:
  $(\textbf{assert}\ ((T_i = 0) \wedge (M_d(d) = false))\ \vee\ ((T_i = 1) \wedge (M_d(d) = true)))$

Figure 5.3: The translation rule for data nodes that are integer and boolean type. The criterial formulas in this translation rule are in blue colour.

As shown in Figure 5.3, a partition switch that has a value of 0, 1 or 2 is created for each data node d that has an integer type. This indicates three different partitions: $> p$, $= p$ and $< p$. If no particular value is given, a value $p = 0$ will be chosen as default. Our technique allows the selection of a single integer to be chosen to divide an integer type attribute into three partitions. These three partitions are directly translated into SMT2 formulas. Similarly, a criterial formula for a boolean data node is formed except that the partition switch is either 0 or 1, since a boolean value can only be *true* and $false$.

5.2.1.1 An Example of Attribute-based Partitions

As an example of how a criterial formula interacts with a data node, we take the metamodel in Figure 5.1. In this metamodel the class called *Class* has an integer type attribute *methodCount* denoting the total number of methods contained in that class. The default strategy for an integer type attribute to achieve 100% is that it uses three partitions: $methodCount < 0$, $methodCount = 0$ and $methodCount > 0$. However, this would conflict with the OCL invariant which requires that *methodCount* must not be a negative number. Thus, with the default strategy only 66.6666% can be achieved. This is because $methodCout < 0$ cannot be chosen by the SMT2 solver. In order to satisfy both OCL invariant and 100% coverage, we use a different strategy. We *manually* choose 3 as the value to partition *methodCount* into the three partitions: $methodCount < 3$, $methodCount = 3$ and $methodCount > 3$.

Given these constraints, the finite universe is formed from which the required instances are derived as follows:

- $V_G = \{class1, class2\}$
- $V_D = \{methodCount1, methodCount2\}$
- $E_{NA} = \{e1 = (class1, methodCount1), e2 = (class2, methodCount2)\}$
- $type_{u,V_G}(class1) = Class,\ type_{u,V_G}(class2) = Class$
- $type_{u,V_D}(methodCount1) = Int,\ type_{u,V_D}(methodCount2) = Int$
- $b(Class) = 2$

Here V_G contains a set of graph nodes and E_{NA} specifies a set of edges that connects graph nodes to data nodes. This finite universe is typed over bounded ATGI by a morphism $type_u$. According to the translation rules described in Chapter 4 (rule 2 in Figure 4.3), every element in V_D is translated into SMT2 constants based on its type. In this example, data nodes $methodCount1$ and $methodCount2$ are translated into integer type SMT2 constants.

Now a partition set $P = \{P_1, P_2\}$ is introduced where the elements P_1 and P_2 correspond to the data nodes $methodCount1$ and $methodCount2$. Two partition switches are created, one each for P_1 and P_2, ranging from 0 to 2 to represent the three possible partitions ($methodCount < 3$, $methodCount = 3$ and $methodCount > 3$). Each partition switch has a one-to-one mapping to a criterial formula. Each mapping is conjoined with a criterial formula which puts an extra constraint on each data node. Thus, the final formula generated is the conjunction of the partition switches for P_1 and P_2, where

$$\begin{aligned} P_1 = &\quad ((T_1 = 0) \wedge (methodCount1 < 3)) \\ &\vee\ ((T_1 = 1) \wedge (methodCount1 = 3)) \\ &\vee\ ((T_1 = 2) \wedge (methodCount1 > 3)). \end{aligned}$$

$$\begin{aligned} P_2 = &\quad ((T_2 = 0) \wedge (methodCount2 < 3)) \\ &\vee\ ((T_2 = 1) \wedge (methodCount2 = 3)) \\ &\vee\ ((T_2 = 2) \wedge (methodCount2 > 3)). \end{aligned}$$

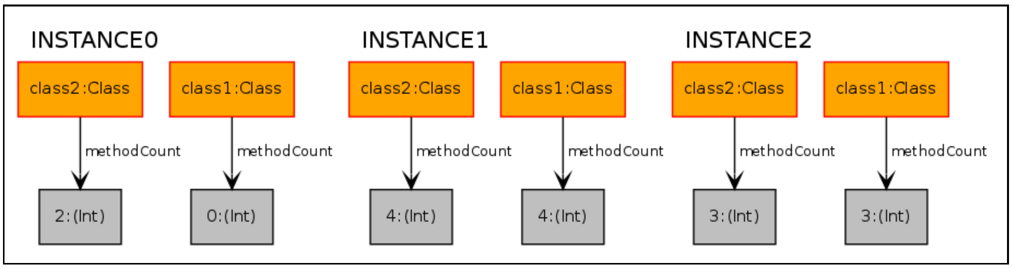


Figure 5.4: Three instances are needed to achieve a maximum *class attribute* coverage for the class *Class* in the metamodel from Figure 5.1.

The disjunction inside P_1 and P_2 makes sure that at least one of the partitions is selected for each instance, and thus at least three instances must be found by the SMT2 solver to achieve 100% coverage for class attribute $methodCount$. Figure 5.4 shows the output generated by our tool in this case, depicting the three instances. These three instances show that the three partitions ($methodCount < 3$, $methodCount = 3$ and $methodCount > 3$) for the integer type attribute $methodCount$ have been covered.

5.2.2 Association-End Multiplicity Partitioning

Associations between classes are an important part of a metamodel, and it is desirable that generated instances should cover these associations in some meaningful way. Andrews et al. have defined an Association-End multiplicity (AEM) coverage criterion for UML class diagrams [Andrews et al., 2003], and this section shows how this can be extended to metamodels and incorporated into a new approach.

To implement AEM coverage, *criterial formulas* (in each translation rule) corresponding to the most frequently used association types defined in a metamodel are specified. These criterial formulas determine how each graph node in an instance are linked to others. In the finite universe (AG_u, $type_u$), all the possible edges from one graph node to another are stored in the set E_G (graph edges). To constrain a particular association, a subset of edges are extracted from E_G and a translation rule is applied.

In order to facilitate the translation to SMT2 formulas, a 2D-array E_{ref} is used to encode the set of edges between two typed graph nodes A and B as described in section 4.2.2. The rows (E_{row}) of E_{ref} encode all links from one instance of A to one instance of B, and the columns (E_{col}) encode all links from one instance of B to one instance of A.

5.2.2.1 Partitioning Unidirectional Associations

Figure 5.5 summarises the translation rules for three different kind of *unidirectional* associations from a class A to a class B. For each association pattern in Figure 5.5, there is a partition switch in each translation rule that is 0 or 1 indicating that the association is divided into two partitions. For example, rule 1 in Figure 5.5 states that when the partition switch T_1 is set to the value 0, no edge encoded by $e_{i,j}$ is chosen, and when T_1 is set to the value 1 only one $e_{i,j}$ is chosen, which indicates that the edge between each instance of A and B is allowed to be presented. The second and third association pattern have a multiplicity of $*$ which indicates that multiple instances of B are allowed to be associated with an instance of A.

The partitions must know the *exact* k number of instances of B that can be associated with an instance of A. To capture this information, we introduce a new auxiliary 2D-array (Aux), where each element in that array is an integer-type SMT2 constant which is called an auxiliary constant $Aux_{i,j}$. Each auxiliary constant can only have a value of either 1 or 0. If $Aux_{i,j}$ is assigned a value of 1, $e_{i,j}$ is then selected. If $Aux_{i,j}$ is assigned a value of 0, $e_{i,j}$ is disabled.

To understand how the auxiliary array in the translation rules works, consider rule 2 as an example. Suppose we assign bounds 3 and 5 to the classes A and B. According to the translation rules described in section 4.2.2, a 2D-array (ref) in Figure 5.7 can be formed to represent all possible links between an instance of A and an instance of B. In the meanwhile, we also create an auxiliary 2D-array (Aux) which has the same size as ref. However, every entry in Aux is an SMT2 integer constant. This array is shown in Figure 5.7. Each $Aux_{i,j}$ in this array can only have a value of either 1 or 0. A value of 1 indicates that a corresponding $e_{i,j}$ from ref is selected otherwise $e_{i,j}$ is not chosen. For illustration purpose, in this example, we choose $k = 3$. Applying translation rule 2, this generates a series of summation equations from each row of Aux, and each equation is either equal to

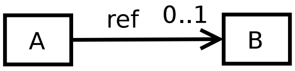
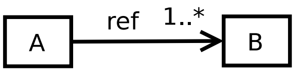
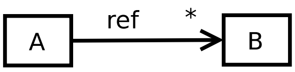
Association Pattern (Unidirectional)	Translation Rule
	(1) $((T_1 = 0) \quad \wedge \quad \bigwedge_{i=1}^{\|E_{row}\|} \bigvee_{j=1}^{\|E_{col}\|} \neg e_{i,j})$ $\vee \quad ((T_1 = 1) \quad \wedge \quad (\bigwedge_{i=1}^{\|E_{row}\|} (\bigvee_{j=1}^{\|E_{col}\|} (\bigwedge_{k=1}^{\|E_{col}\|} k \neq j \rightarrow \neg e_{i,k}) \wedge e_{i,j})))$
	(2) $((T_1 = 0) \quad \wedge \quad \bigwedge_{i=1}^{\|E_{row}\|} (\sum_{j=1}^{\|E_{col}\|} Aux_{i,j}) = 1)$ $\vee \quad ((T_1 = 1) \quad \wedge \quad \bigwedge_{i=1}^{\|E_{row}\|} (\sum_{j=1}^{\|E_{col}\|} Aux_{i,j}) = k \quad \wedge \quad \|E_{col}\| > 1)$ where $1 < k \leq \|E_{col}\|$
	(3) $((T_1 = 0) \quad \wedge \quad \bigwedge_{i=1}^{\|E_{row}\|} \bigvee_{j=1}^{\|E_{col}\|} \neg e_{i,j})$ $\vee \quad ((T_1 = 1) \quad \wedge \quad \bigwedge_{i=1}^{\|E_{row}\|} (\sum_{j=1}^{\|E_{col}\|} Aux_{i,j}) = k \quad \wedge \quad \|E_{col}\| \geq 1)$ where $1 \leq k \leq \|E_{col}\|$

Figure 5.5: Translation rules for graph edges typed over the edges in $ATGI_b$ that represent unidirectional associations. The criterial formulas in translation rule are marked in blue colour.

$$ref = \begin{array}{c|ccccc} & b_1 & b_2 & b_3 & b_4 & b_5 \\ \hline a_1 & e_{1,1} & e_{1,2} & e_{1,3} & e_{1,4} & e_{1,5} \\ a_2 & e_{2,1} & e_{2,2} & e_{2,3} & e_{2,4} & e_{2,5} \\ a_3 & e_{3,1} & e_{3,2} & e_{3,3} & e_{3,4} & e_{3,5} \end{array}$$

Figure 5.6: The 2D-array representing the second association pattern in Figure 5.5.

$$Aux = \begin{array}{c|ccccc} & b_1 & b_2 & b_3 & b_4 & b_5 \\ \hline a_1 & Aux_{1,1} & Aux_{1,2} & Aux_{1,3} & Aux_{1,4} & Aux_{1,5} \\ a_2 & Aux_{2,1} & Aux_{2,2} & Aux_{2,3} & Aux_{2,4} & Aux_{2,5} \\ a_3 & Aux_{3,1} & Aux_{3,2} & Aux_{3,3} & Aux_{3,4} & Aux_{3,5} \end{array}$$

Figure 5.7: The auxiliary 2D-array created for capturing information about the exact number of instances of B that an instance of A can be associated with. Each $Aux_{i,j}$ is an SMT2 integer constant.

1 or 3. For example, Figure 5.8 shows the resulting formula from the first row of ref. Thus, translation rule 2 precisely captures the meaning of two partitions: each instance of A is associated with exactly one instance of B and each instance of A is associated with multiple instances of B.

Figure 5.9 shows a possible assignment for each entry in the Aux array. The summation from each row in Aux here is 3. This indicates that each instance of A is associated with exactly three instances of B. This information is reflected back to the original 2D-array (ref). For example, if $Aux_{1,2}$ is assigned to a value of 1, then $e_{1,2}$ in ref is also selected. This represents a link between a_1 and b_2 is established.

Therefore, each auxiliary array (Aux) used in the translation rules in Figure 5.5 controls the exact k number of the instances of B associated with an instance of A by writing the formula as a summation of a selection of values in $Aux_{i,j}$s. To cover a partition, the upper bound of k is constrained by $|E_{col}|$, the bound on the number of instances of B. We also allow users to constrain any number between 1 and E_{col} for other usages such as generating instances that have a specific number

$$\begin{array}{l} ((Aux_{1,1} + Aux_{1,2} + Aux_{1,3} + Aux_{1,4} + Aux_{1,5}) = 1) \\ \vee((Aux_{1,1} + Aux_{1,2} + Aux_{1,3} + Aux_{1,4} + Aux_{1,5}) = 3) \end{array}$$

Figure 5.8: The resulting formula for the first row of the array ref.

$$Aux = \begin{array}{c|ccccc} & b_1 & b_2 & b_3 & b_4 & b_5 \\ \hline a_1 & 0 & 1 & 0 & 1 & 1 \\ a_2 & 1 & 0 & 1 & 0 & 1 \\ a_3 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Figure 5.9: An example of a possible assignment found by an SMT2 solver for array ref in Figure 5.7.

of links. Thus, rule 3 in Figure 5.5 also captures two partitions: each instance of A is associated with zero instance of B and each instance of A is associated with one or more instances of B.

5.2.2.2 Partitioning Bidirectional Associations

The translation rules used for bidirectional association are similar to those for unidirectional associations except that the maximum possible number of instances of B that each instance of A can connect to, must be calculated. Figure 5.10 summarises the translation rules for the most commonly used *bidirectional* associations. The partition switch T_1 in each translation rule selects between two criterial formulas indicating that each bidirectional association pattern is partitioned into two.

The first and second criterial formulas for the first association pattern (1:0..1)[2] specifies that either no instances or one instance of B is connected to an instance of A. The first criterial formula for the second association pattern (1:1..$*$) is similar to the one used for unidirectional associations. However, for the second criterial formula the maximum possible number of instances of B that an instance of A can connect to needs to be calculated. That is the upper bound of this number k. This is computed by calculating the difference between the bound of B and the bound of A, and adding 1. Since $|E_{row}|$ specifies the bound of A while $|E_{col}|$ gives the bound of B, to understand how to calculate the upper bound of k, we consider the following three scenarios:

- $|E_{col}| = |E_{row}|$, meaning that we have an equal number of instances of A and B. Since the multiplicities for two association-ends (1 and 1..$*$) state

[2] We use x:y to denote the multiplicities x and y at two association-ends.

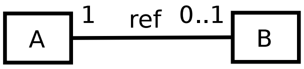
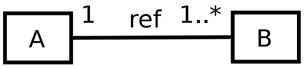
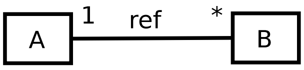
Association Pattern (Bidirectional)	Translation Rule
	(1) $((T_1 = 0) \wedge \bigwedge_{i=1}^{\lvert E_{row}\rvert} \bigvee_{j=1}^{\lvert E_{col}\rvert} \neg e_{i,j})$ $\vee \quad ((T_1 = 1) \wedge \bigvee_{i=1}^{\lvert E_{row}\rvert} (\sum_{j=1}^{\lvert E_{col}\rvert} Aux_{i,j}) = 1)$
	(2) $((T_1 = 0) \wedge \bigvee_{i=1}^{\lvert E_{row}\rvert} (\sum_{j=1}^{\lvert E_{col}\rvert} Aux_{i,j}) = 1)$ $\vee \quad ((T_1 = 1) \wedge \bigvee_{i=1}^{\lvert E_{row}\rvert} (\sum_{j=1}^{\lvert E_{col}\rvert} Aux_{i,j}) = k) \wedge \lvert E_{row}\rvert < \lvert E_{col}\rvert$ where $1 < k \leq \lvert E_{col}\rvert - \lvert E_{row}\rvert + 1$
	(3) $((T_1 = 0) \wedge \bigwedge_{i=1}^{\lvert E_{row}\rvert} \bigvee_{j=1}^{\lvert E_{col}\rvert} \neg e_{i,j})$ $\vee \quad ((T_1 = 1) \wedge \bigvee_{i=1}^{\lvert E_{row}\rvert} (\sum_{j=1}^{\lvert E_{col}\rvert} Aux_{i,j}) = k) \wedge \lvert E_{row}\rvert \leq \lvert E_{col}\rvert$ where $1 \leq k \leq \lvert E_{col}\rvert - \lvert E_{row}\rvert + 1$

Figure 5.10: Translation rules for graph edges typed over the edges in $ATGI_b$ that represent bidirectional associations. The criterial formulas in each translation rule are marked in blue colour.

that one instance of A must be connected to at least one instance of B, and one instance of B can only be linked to one instance of A, this scenario now implies that each instance of A connects each instance of B, and vice versa. Thus, the maximum number of instances of B that an instance of A can connect to is $k = 1$.

- $|E_{col}| > |E_{row}|$, meaning that we have more instances of B than A. Every instance of A is first connected to one instance of B, and every instance of B connects to only one instance of A. Now, the remaining number of instances of B is added to one of the existing connections between an instance of A and and instance of B, where any link that has already been established must be counted ($k = |E_{col}| - |E_{row}| + 1$). Thus, k gives the maximum number of B's that one of the instances of A can connect to.

- $|E_{col}| < |E_{row}|$, meaning that we have less instances of B than A. How-

$$Aux = \begin{array}{c|ccccc} & b_1 & b_2 & b_3 & b_4 & b_5 \\ \hline a_1 & 1 & 0 & 0 & 1 & 1 \\ a_2 & 0 & 0 & 1 & 0 & 0 \\ a_3 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Figure 5.11: An example of an assignment found by an SMT2 solver for an association between two classes A and B, where the bounds for A and B are manually allocated and they are 3 and 5, respectively. In this example an instance of A is linked with a maximum of 3 instances of B. Since the sum of each column is 1, an instance of B is only connected to a single instance of A.

ever, this scenario violates the constraint implied by the association-end multiplicities, and is thus ruled out.

Thus, in both possible cases the minimum number of instances of B has to be equal to the number instances of A. Figure 5.11 shows an example where one instance of A can connect to at most 3 instances of B. The bounds for A and B is manually allocated here.

The third association pattern (1:$*$) in Figure 5.10 has a criterial formula that combines the first criterial formula from the first association-pattern with the second criterial formula from the second association-pattern. Users can specify the exact number of links for an association between 1 and $|E_{col}| - |E_{row}| + 1$, since the lower bound for this association is captured by the first sub-formula in rule 3.

5.2.3 Better Control of Instance Enumeration

With respect to the coverage criteria presented in previous sections, we devise a way to reduce the number of instances during the enumeration. We first group partition switches that have the same number of partitions, and then apply Formula 5.1 to block unnecessary assignments. To group them together, we store every partition switch in a hash table, indexed by their number of partitions. Then we apply Formula 5.1 on each group of switches. The first sub-formula in Formula 5.1 indicates that every partition switch that has the same number of partitions must be equal to each other. This allows us to achieve the coverage criteria partitions by partitions. The second sub-formula indicates that it is also

2	T_1, T_2, T_3, T_4
3	T_5, T_6

Table 5.1: Partition switches are grouped by the number of partitions

possible for switches that have the same number of partitions to have different combinations. This allows us to achieve the coverage criteria using a combination of partitions. Finally, we conjoin all the formulas from each group to have a combination of partition switches that have different partitions.

$$(\bigwedge_{i=1}^{n-1} T_i = T_{i+1}) \quad \vee \quad (\bigwedge_{i=1}^{n-1} T_i \neq T_{i+1}),\ where\ n > 2. \tag{5.1}$$

To understand how this works, please consider the following example:

Suppose a total of six partition switches ($T_1, T_2, T_3, T_4, T_5, T_6$) were created, four of them (T_1, T_2, T_3, T_4) have two partitions and two of them (T_5, T_6) have three partitions. We first group those six partition switches, as in Table 5.1, by their partitions. Then we apply Formula 5.1 to these two groups. This results in the following formulas:

$$f_1 = ((T_1 = T_2) \wedge (T_2 = T_3) \wedge (T_3 = T_4)) \vee ((T_1 \neq T_2) \wedge (T_2 \neq T_3) \wedge (T_3 \neq T_4))$$

$$f_2 = (T_5 = T_6) \vee (T_5 \neq T_6)$$ [3]

We observe these formulas, not every assignment that makes f_1 and f_2 *satisfiable* is possible now. For example, T_1, T_2, T_3, T_4 can have a total of sixteen possible assignments ($T_1 \vee T_2 \vee T_3 \vee T_4$). However, after applying Formula 5.1, only four possible assignments are left. These four possible assignments are listed in Table 5.3. The first two assignments satisfy the first sub-formula of Formula 5.1, and the remaining two satisfy the second sub-formula. Thus, by applying Formula 5.1, we block twelve other possible assignments. Furthermore, the first two assignments achieve full coverage via partitions. The last two assignments achieve full coverage via a combination of partitions. Therefore, by enumerating these four possible assignments, we can cover all the partitions for partition switches (T_1,T_2,T_3 and T_4) in two different ways.

[3] Note that when there are only two partition switches (T_i and T_j), we use formula $(T_i = T_j) \vee (T_i \neq T_j)$.

T_1	T_2	T_3	T_4
1	1	1	1
2	2	2	2
1	2	1	2
2	1	2	1

Table 5.2: Four possible assignments for formula f_1. Here, 1 and 2 denote two different partitions.

T_1	T_2	T_3	T_4	T_5	T_6
1	2	1	2	3	2

Table 5.3: One possible assignment for formula f_1 and f_2. Here, 1, 2 and 3 denote three different partitions.

Finally, in order to have a different combination from partition switches that have different partitions, we conjoin the two formulas ($f_1 \wedge f_2$) and feed the entire formula to an SMT2 solver. For example, Table 5.3 shows one possible assignment for all partition switches ($T_1, T_2, T_3, T_4, T_5, T_6$).

5.3 An Example of Achieving CA and AEM Coverage Criteria

To demonstrate partition-based generation, we use a subset of the Ecore metamodel as depicted in Figure 5.12 [Steinberg et al., 2008]. This metamodel describes the relationships among *EPackage*, *EClass*, *EAttribute*, and *EOperation* in the Ecore metamodel. The bound for each non-abstract class is shown as a number in a circle.

In order to achieve the maximum coverage for both CA and AEM, the translation rules described in section 5.2 are applied to this metamodel, and result in a total of 8 instances. Thus, one can conclude that only 8 instances are needed to achieve the full coverage of CA and AEM for this metamodel. These 8 instances cover a range of combinations from different partition switches defined during the translation to SMT2 formulas. For example, two of the instances in Figure 5.13 and 5.14 show a combination of different partitions from different associations and attributes defined in the metamodel.

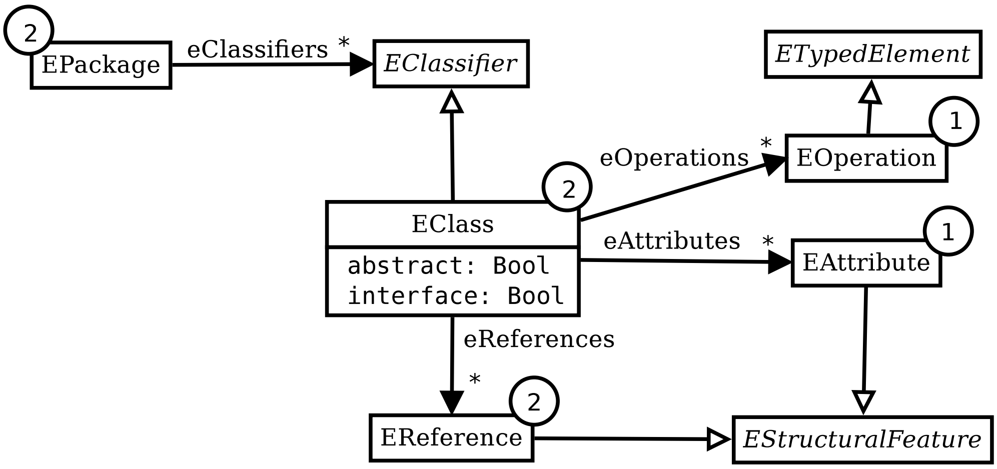


Figure 5.12: A subset of the Ecore metamodel showing the relationship between *EPackage*, *EClass*, *EAttribute*, *EReference* and *EOperation*, where a bound for each non-abstract class is depicted as a number in a circle.

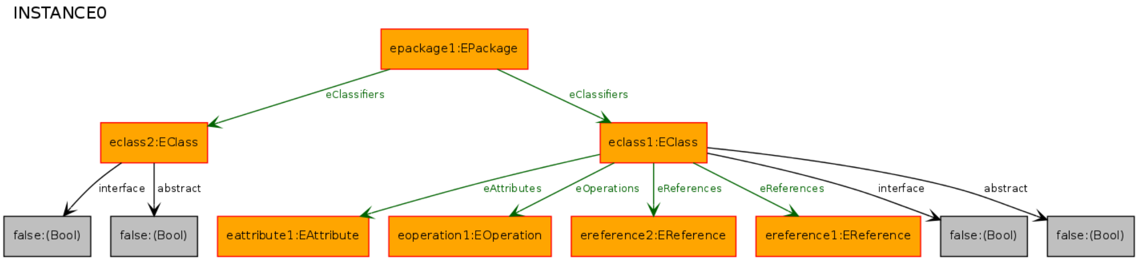


Figure 5.13: In this generated instance of the Ecore metamodel of Figure 5.12, at least one *EClass* instance is associated with a maximum number of *EAttribute*, *EOperation* and *EReference* instances.

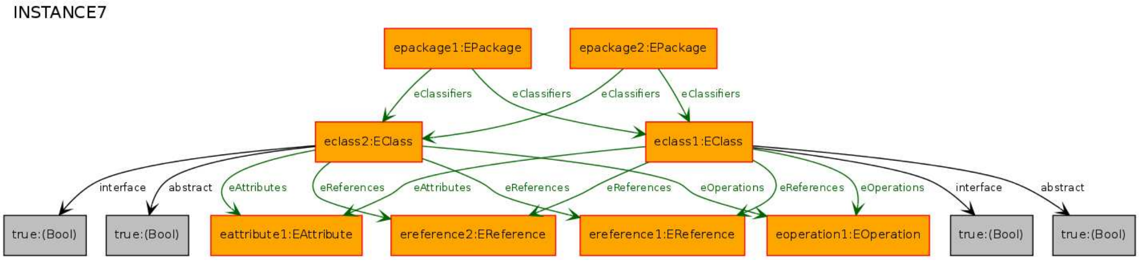


Figure 5.14: In this generated instance of the Ecore metamodel of Figure 5.12, each $EClass$ instance is associated with a maximum number of other instances ($EAttribute$, $EOperation$ and $EReference$) according to the bound defined on each class in metamodel.

5.4 Evaluation

To effectively evaluate the technique described in this chapter, we have implemented it in our tool (ASMIG), and a list of metamodels have been collected for testing purpose. This section presents the results from our evaluation for ASMIG.

5.4.1 Implementation

The ASMIG tool has been extended by adding a new component that caches a set of formulas representing associations in a metamodel (as can be seen in Figure 1.4). The cached formulas are used when ASMIG generates the partition switches and criterial formulas for each translation rule. Finally, the newly generated formulas (partition switches and criterial formulas) are conjoined with the formulas described in Chapter 4, and input to the SMT2 solver.

5.4.2 Results

The evaluation for ASMIG for partition criteria are conducted on a machine with a Intel Core 2 Duo (E7500) CPU 2.93GHz, and a 4GB memory. The results are shown in Table 5.4. We use the same set of metamodels from Table 4.1 except that we replaced two of them (C and DoDAF-SV5) with Ant and HTML as most of the associations in C and DoDAF-SV5 metamodels are 1:1 which are not helpful for testing our partition-based criteria instance generation. To apply ASMIG to these metamodels, an appropriate bound for each class in a metamodel is calculated. The total bounds, as listed in Table 5.4 indicates the sum of each bound allocated for each non-abstract class of a metamodel. The bound for each class is *automatically* calculated by using linear integer arithmetic theory. 0 is *manually* chosen as a default value for three partitions (< 0,$= 0$ and > 0) for each integer type class attribute. For the Association-End Multiplicity criterion (AEM), a minimum of 3 connected instances is chosen to show multiple connections for an association that have a $*$ as one of the association ends. We choose 3 because we consider it can give us a relatively reasonable size of a graph and it can also facilitate anyone to validate visually.

To analyse the factors that affect the translation time, we use a similar approach as described in section 4.5.2. We define the size of a metamodel as the number of classes in a metamodel. We define it in this way because a metamodel with a larger number of classes is more likely to have a complex structure. We then plot a graph for translation time against different sized metamodels from Table 5.4 in Figure 5.15.

Through analysing Figure 5.15 and Table 5.4 together, we note that the translation time is not affected by a single factor. In fact, it is affected by three factors: the size of a metamodel, the number of associations and the number of attributes. A smaller metamodel with more associations and attributes, ASMIG might need more time for translation than a larger metamodel with less associations and attributes. This is because each association and attribute needs to be additionally constrained in order to achieve partition-based coverage criterion. For example, the metamodel $HTML$ has more classes than the $UML2$ *Class Diagram* metamodel, but ASMIG spends much more time on translating the $UML2$ *Class Diagram* metamodel than $HTML$ because the $UML2$ *Class Diagram* has 26 associations and 46 attributes while $HTML$ only has 7 associations and no attributes. Therefore, translation time from a metamodel to formulas depends on its size, associations and attributes.

The average finding time indicates average instance enumeration for each metamodel and it mainly depends on the total bounds defined for a metamodel. The total instances indicates the number of instances needed for achieving full coverage for CA and AEM criteria. As can be seen, the metamodels with no attributes are covered by 4 instances. This is because the associations of those metamodels are partitioned into two partitions, each partition is covered by one instance, plus extra two instances for a combination of partitions (as this is shown in the example in section 5.2.3). The metamodels with attributes need more instances to achieve a full CA and AEM coverage separately, and a combination of both. All instances generated for each metamodel in Table 5.4 are available at our website[4].

[4] http://www.cs.nuim.ie/~haowu/ASMIG/Results/PartitionBased

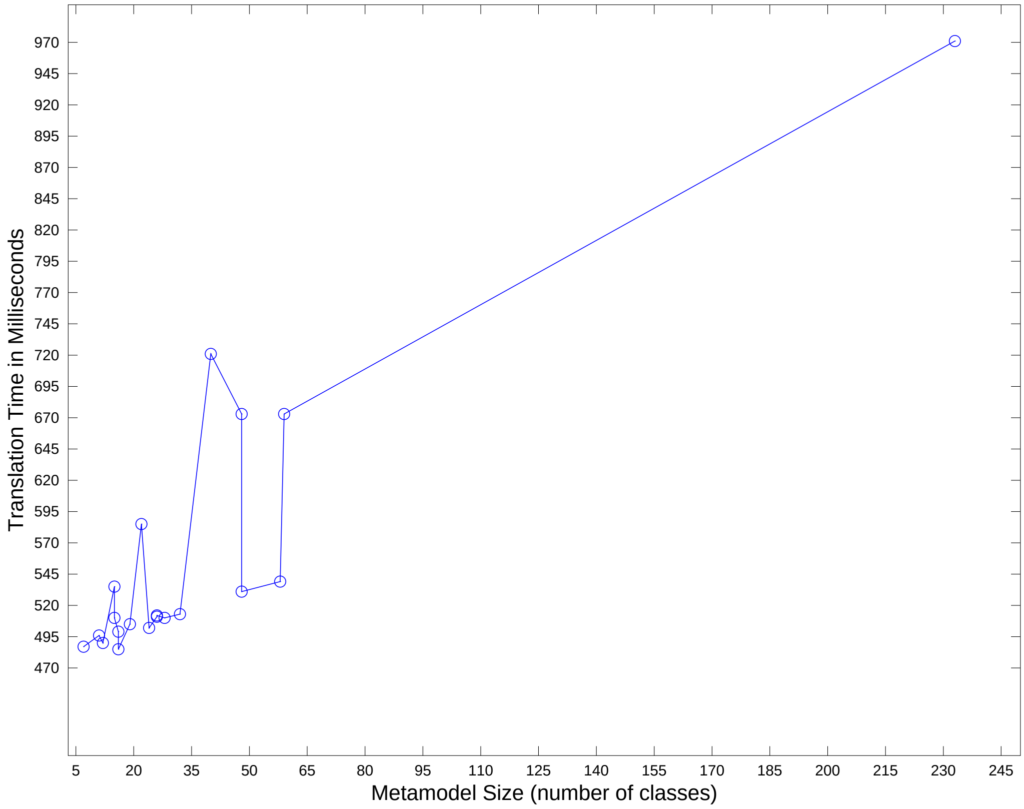


Figure 5.15: Translation time affected by different sized metamodel in Table 5.4. Each point in this graph represents the translation time that ASMIG takes on a specific metamodel from Table 5.4.

	Number of			Total		Time in ms	
Metamodel	Classes	Assocs	Attribs	Bounds	Instances	Translation	Avg Finding
Ant [5]	48	27	0	56	4	673ms	71ms
Company [6]	7	6	6	13	12	487ms	36ms
C++ 1.0 [5]	16	4	5	20	8	499ms	22ms
Java [9]	233	104	1	183	8	971ms	2629ms
Royal&Loyal [7]	15	41	2	40	8	535ms	65ms
Finite State Machine 1.0 [5]	16	7	0	16	4	485ms	30ms
Ecore [8]	22	40	0	31	4	585ms	195ms
UML2 Class Diagram [8]	40	26	46	35	8	721ms	966ms
Web App: Conceptual Model[5]	19	24	0	25	4	505ms	45ms
KM3 [5]	12	7	0	16	4	490ms	40ms
Business Process Model [5]	26	15	0	28	4	511ms	63ms
CPL1.0 [5]	32	16	0	38	4	513ms	90ms
GraphML [5]	11	13	2	20	8	496ms	53ms
Hierarchical State Machine 1.0 [5]	15	16	0	33	4	510ms	72ms
Maven(maven.xml) 0.3 [5]	58	32	0	65	4	539ms	108ms
MoDAF0.1 [5]	48	35	0	70	4	531ms	76ms
QualityofService [5]	24	26	0	37	4	502ms	69ms
DOT1.0 [5]	26	20	0	21	4	512ms	50ms
BibTexML1.2 [5]	28	4	0	18	4	510ms	28ms
HTML [5]	59	7	0	59	4	673ms	1562ms

Table 5.4: Results of 20 metamodels for evaluating partition-based instance generation, and all instances were automatically generated by the ASMIG tool.

5.5 Summary

This chapter describes our unique technique for generating instances that achieves partition-based criteria. This technique introduces the concepts of a switch partition and criterial formula. A set of translation rules have been developed based on these two concepts. The partition switch controls which particular partitions are selected, and a criterial formula manages the instance corresponds to a particular partition. The translation rules are successfully implemented into a ASMIG tool, and evaluation results show the feasibility of this technique on different size of metamodels.

[5] available at: http://www.emn.fr/z-info/atlanmod/index.php/Ecore

[6] simple example similar to Figure 4.2, available at website

[7] from Eclipse Modeling Framework Royal and Loyal Example Project

[8] extracted from Eclipse Modeling Framework

[9] available at: http://www.jamopp.org/index.php/JaMoPP_Download

Chapter 6

Generating Metamodel Instances Satisfying Graph-Based Criteria

The technique presented in the previous chapter will help testers to enumerate instances that contribute to partition based coverage. However, partition-based instance generation is not enough, as a tester may seek an instance that can be used beyond coverage contribution. In particular, testers may want instances (programs) from a general purpose or domain specific metamodel, and use these instances for testing tools such as compilers, formatters, refactoring tools, metrics calculators, etc.

For example, consider the metamodel in Figure 6.1. This presents a subset of a Java programming language metamodel capturing the language features *class*, *method* and $fields$. [Heidenreich et al., 2010a]. A tester may require a metamodel instance generator to generate a valid Java program with a particular call depth for testing Java compilers or a particular value of a cohesion metric for testing metric calculators.

Generating this type of instance is the same as generating instances that have specific graph properties, since we represent our metamodel in a bounded graph ($ATGI_b$). Thus, in order to achieve such instance generation, we present another unique technique that encodes common graph-based properties into SMT2 formulas according to different scenarios. With this technique, it is possible to generate metamodel instances beyond coverage contribution. This technique has already

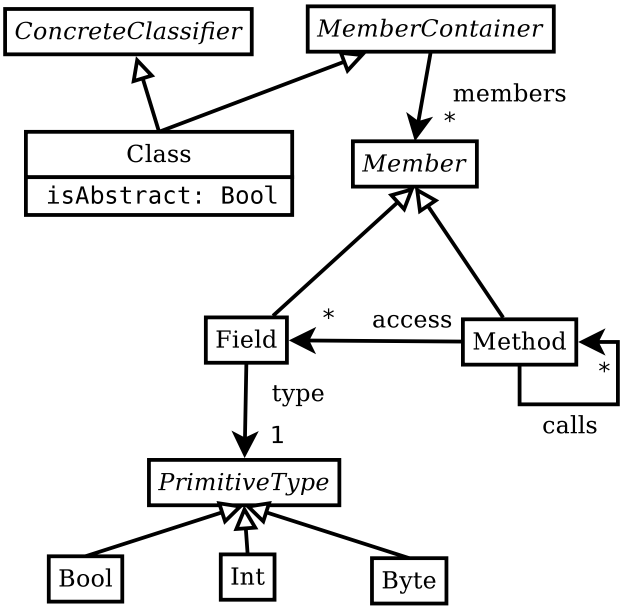


Figure 6.1: A subset of the Java metamodel representing relationships between classes and their members, which are fields or methods.

been implemented into our ASMIG tool, and our evaluation of the ASMIG tool demonstrates the feasibility of this technique.

Recall that we represent a metamodel as a bounded graph ($ATGI_b$) and translate the finite universe (AG_u) to SMT2 formulas via a set of translation rules described in sections 4.2.1 and 4.2.2. This includes our encoding of possible links for a particular association into a 2D-array (every entry is an SMT2 boolean constant). It is possible to manipulate this 2D-array to form new formulas that can express how graph nodes are connected to each other. Note that formulas 6.1 through 6.5, described in this chapter are translation rules for encoding graph-based properties according to different scenarios.

6.1 Directed Acyclic Graphs

Directed acyclic graphs (DAGs) are commonly used in many areas, for example, the topology of a network, data flow diagrams, etc. Regarding metamodeling, one may require a program to have a particular depth of inheritance tree, or a particular call depth. Thus, to ensure the generation of a DAG from a reflexive

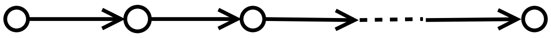


Figure 6.2: A DAG with all nodes in one line.

association in a metamodel, we only enable the elements that are in the upper triangle of the 2D-array, and disable the rest of the $e_{i,j}s$ in the 2D-array, breaking all the cycles in the graph. Our approach is similar to the one used in [Shlyakhter, 2007], but differs by the addition of quantitative constraints, as expressed in the following formula:

$$(\bigvee_{i=1}^{|Q|} Q_i) \wedge \ (\bigwedge_{i=1}^{|E_{row}|-1} ((\bigvee_{j=1}^{|E_{col}|} e_{i,j}) \rightarrow (\bigwedge_{k=i+1}^{|E_{row}|} \neg e_{k,j}))) \tag{6.1}$$

The longest path length that a DAG (contains at least one node) can contain is never greater than $b(N) - 1$, where N is a graph node. To see why this is $b(N) - 1$:

Suppose we have a DAG with $|V|$ nodes, where V is the set of nodes and $|V| \geq 1$. The longest path length varies between 0 and $|V| - 1$. Because for a fully connected DAG, one can always organise the nodes from this DAG into a line shape. All nodes are connected and the edges between two nodes going to the same direction. This also illustrates with the Figure 6.2. We know the number of nodes can be calculated from bound function $b(N)$. Therefore, $b(N) - 1$ is the longest path length a DAG can *possibly* contain.

Since the number of elements in the upper triangle of the 2D-array is finite, we compute every possible path that has this particular length by iterating through every element in the upper triangle of the 2D-array, and save them into a set Q. Each element (Q_i) in Q is a conjunction over a subset of edges extracted from 2D-array, and it encodes a possible path to be selected by the SMT2 solver. Thus, the disjunction in Formula 6.1 ensures the selection of at least one possible path from the set Q.

To guarantee that every graph found contains a particular path length, we add an extra formula specifying that when some elements from a row in the triangle are selected then nothing from the rows below can be chosen. As before, E_{row} and E_{col} in formula 6.1 are used to capture the rows and columns of the upper-triangular 2D-array.

6.2 Sharing and Non-Sharing Nodes

The association multiplicities defined on a metamodel constrains how the nodes are linked. However, a user may wish to specify the connectivity of each node in a metamodel instance to achieve a particular shaped graph for some situations such as measuring cohesion metrics. In order to facilitate instance generation with such properties, we introduce the following new properties.

In a graph, some nodes can have their out-going edges all going to the same nodes and some do not. We consider these nodes having sharing and non-sharing properties. Sharing and non-sharing properties can only be applied to a *non-reflexive* association. Before we precisely define sharing and non-sharing properties, we first define two functions (f and g).

1. Function f is an out-adjacency function (Adj^{+}) that computes a set of graph nodes from all out-going edges of a particular graph node. $f : V_G \to 2^{V_G}$, where V_G is the set of graph nodes, and 2^{V_G} is the power set of V_G.

2. Function g is an in-adjacency function (Adj^{-}) that computes a set of graph nodes from all in-coming edges of a particular graph node: $g : V_G \to 2^{V_G}$, where V_G is the set of graph nodes, and 2^{V_G} is the power set of V_G.

With functions f and g, we are able to calculate a set of nodes based on their in-coming and out-going edges. Now we can use these two functions to define the following sharing and non-sharing properties:

- A set of graph nodes $S = \{N_1, N_2, ..., N_j\}$, where $|S| \geq 2$ are said to be *strong sharing nodes* iff $(\bigcap\limits_{i=1}^{j} f(N_i)) \neq \emptyset$, $\forall S_x \in \bigcup\limits_{i=1}^{j} f(N_i)$ and $g(S_x) \subseteq S$.

- A set of graph nodes $S = \{N_1, N_2, ..., N_j\}$, where $|S| \geq 2$ are said to be *weak sharing nodes* iff $(\bigcap\limits_{i=1}^{j} f(N_i)) \neq \emptyset$, $\exists S_x \in \bigcup\limits_{i=1}^{j} f(N_i)$ and $S \subset g(S_x)$.

- A set of graph nodes $S = \{N_1, N_2, ..., N_j\}$, where $|S| \geq 2$ are said to be *strong non-sharing nodes* iff $\forall N_i \in S$, $|f(N_i)| = 1$ and $f(N_a) \cap f(N_b) = \emptyset$, where $1 \leq a < b \leq j$.

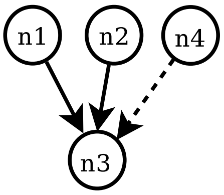


Figure 6.3: An example of sharing nodes in a graph

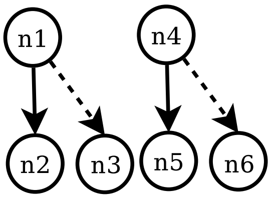


Figure 6.4: An example of non-sharing nodes in a graph

- A set of graph nodes $S = \{N_1, N_2, ..., N_j\}$, where $|S| \geq 2$ are said to be *weak non-sharing nodes* iff $\forall N_i \in S$, $|f(N_i)| > 1$ and $f(N_a) \cap f(N_b) = \emptyset$, where $1 \leq a < b \leq j$.

To understand these concepts, we use two examples to illustrate sharing and non-sharing properties. In Figure 6.3, a solid line is used to denote the existing links and a dashed line is used to represent possible links. The set of nodes $n1$ and $n2$ (with solid lines) are considered as strong sharing nodes since their both out-adjacency functions return $n3$ ($f(n1) = f(n2) = \{n3\}$), and $n3$'s in-adjacency function returns $n1$ and $n2$ ($g(n3) = \{n1, n2\}$) . In other words, $n3$ can only be accessed by both $n1$ and $n2$ and no other nodes. However, if a link from $n4$ to $n3$ is connected, then the set of nodes $n1$ and $n2$ are regarded as weak sharing nodes because $n3$'s in-adjacency function this time returns three nodes: $g(n3) = \{n1, n2, n4\}$. Thus, the set of nodes $n1$, $n2$ and $n4$ are considered as strong sharing nodes ($f(n1) \cap f(n2) \cap f(n4) = n3$), and $g(n3) \subseteq \{n1, n2, n4\}$).

Similarly, in Figure 6.4 the solid lines between nodes $n1$, $n2$ and $n4$, $n5$ make the set of nodes $n1$ and $n4$ strong non-sharing nodes in the graph ($|f(n1)| = |f(n4)| = 1$, and $f(n1) \cap f(n4) = \emptyset$). If $n1$ also connects to $n3$ (a possible link), and $n4$ connects to $n6$, then the set of nodes $n1$ and $n2$ are weak non-sharing nodes, since they all connect to more than one other node ($|f(n1)| = |f(n4)| > 1$).

Since all possible links are encoded in the 2D-array (as described in section

	a_1	a_2	a_3	a_4
b_1	$e_{1,1}$	$e_{1,2}$	$e_{1,3}$	$e_{1,4}$
b_2	$e_{2,1}$	$e_{2,2}$	$e_{2,3}$	$e_{2,4}$
b_3	$e_{3,1}$	$e_{3,2}$	$e_{3,3}$	$e_{3,4}$

Figure 6.5: An example of transposed 2D-array for illustrating sharing and non-sharing properties.

4.2.2) and each element in the 2D-array is an SMT2 constant that encodes an edge, sharing and non-sharing nodes indicate that all the edges encoded by SMT2 constants in the array may have out-going edges going to the same set of nodes (encoded by SMT2 constants) or not. In order to help readers visualise these properties, we first transpose our 2D-array and interpret the column of the 2D-array as all out-going edges from one particular graph node to other nodes. For example, Figure 6.5 shows an array has been transposed with columns that denote all out-going edges from $a's$ to $b's$. For example, in column 1, $e_{1,1}$ indicates an out-going edge from a_1 to b_1.

Formulas 6.2 to 6.5 are used to constrain the 2D-array in order to get a specific sharing or non-sharing property. In each formula, the set L contains a list of column numbers of the 2D-array. We use this L to denote a set of graph nodes to be assigned with one of the four properties, and L_k denotes the kth element in this set. For example, $L = \{1\}$ indicates that the first element (L_1) specifies a graph node in the first column from 2D-array, and that is graph node a_1 in Figure 6.5. Additionally, we require that $|L| \geq 2$ because we need sharing and non-sharing property can only be applied to at least two nodes.

Formula 6.2 specifies the strong sharing property. The first sub-formula indicates that a set of nodes must share some common nodes by listing all possible choices while the second sub-formula turns off all other nodes which are not specified in L. To understand how this works, we use the array in Figure 6.5 as an example. Suppose we want to give strong sharing property to nodes: a_1 and a_4 ($L = \{1, 4\}$). This indicates that at least one of the $b's$ must be shared by them. For example, $e_{1,1}$ and $e_{1,4}$ could be selected at the same time *or* $e_{2,1}$ and $e_{2,4}$ are chosen $((e_{1,1} \wedge e_{1,4}) \vee (e_{2,1} \wedge e_{2,4}))$. This represents that a_1 and a_4 they both have out-going edges to b_1 or b_2. This is captured by the first sub-formula in Formula

6.2. Now, suppose $e_{1,1}$ and $e_{1,4}$ are selected, then anything between them cannot be selected otherwise they are not strong sharing nodes. Thus, $e_{1,2}$ and $e_{1,3}$ are disabled when $e_{1,1}$ and $e_{1,4}$ are selected ($(e_{1,1} \wedge e_{1,4}) \rightarrow (\neg e_{1,2} \wedge \neg e_{1,3})$). This is captured by the second sub-formula in Formula 6.2.

For the weak sharing property, Formula 6.3 is similar to Formula 6.2 except that we drop the second sub-formula. Instead, we add a formula that states that at least one of the $a's$ not specified in L can be linked to the $b's$. Considering the example in Figure 6.5, when $e_{1,1}$ and $e_{1,4}$ are both selected, $e_{1,2}$ or $e_{1,3}$ can possibly be selected as well ($(e_{1,1} \wedge e_{1,4}) \wedge (e_{1,2} \vee e_{1,3})$). This means that a graph node can also be shared by other nodes.

$$(\bigvee_{i=1}^{|E_{row}|} \bigwedge_{k=1}^{|L|} e_{i,L_k}) \quad \wedge \quad (\bigwedge_{i=1}^{|E_{row}|} (\bigwedge_{k=1}^{|L|} e_{i,L_k} \rightarrow \bigwedge_{j=1, j \notin L}^{|E_{col}|} \neg e_{i,j})) \tag{6.2}$$

$$\bigvee_{i=1}^{|E_{row}|} \bigwedge_{k=1}^{|L|} e_{i,L_k} \wedge \bigwedge_{i=1}^{|E_{row}|} \bigvee_{j=1, j \notin L}^{|E_{col}|} e_{i,j} \tag{6.3}$$

A similar approach is used for generating strong and weak *non*-sharing nodes in the graph, and these constraints are encoded in Formula 6.4 and 6.5 respectively.

$$(\bigwedge_{k=1}^{|L|} \bigvee_{i=1}^{|E_{row}|} (\bigwedge_{j=1}^{|E_{row}|} j \neq i \rightarrow \neg e_{j,k}) \wedge e_{i,L_k}) \wedge \quad (\bigwedge_{i=1}^{|E_{row}|} (\bigwedge_{k=1}^{|L|} e_{i,L_k} \rightarrow \bigwedge_{j=1, j \notin L}^{|E_{col}|} \neg e_{i,j})) \tag{6.4}$$

$$(\bigwedge_{k=1}^{|L|} \bigvee_{i=1}^{|E_{row}|} e_{i,L_k}) \quad \wedge \quad (\bigwedge_{i=1}^{|E_{row}|} (\bigwedge_{k=1}^{|L|} e_{i,L_k} \rightarrow \bigwedge_{j=1, j \notin L}^{|E_{col}|} \neg e_{i,j})) \tag{6.5}$$

Formula 6.4 indicates that only one edge can be selected from a corresponding column. Again, here we store all corresponding column numbers in the set L. It is easy to see that as long as one edge is selected all other edges in the same row and column are switched off. For example, if we select $e_{1,1}$ from the array in Figure 6.5, then $e_{2,1}$ and $e_{3,1}$ are disabled. This is captured by the first sub-formula. Meanwhile, $e_{1,2}$, $e_{1,3}$ and $e_{1,4}$ are also disabled. This is captured by the second sub-formula. Thus, this is precisely saying that b_1 can only be connected to a_1.

Similarly, for weak non-sharing property, the first sub-formula in Formula 6.5 indicates that there could be multiple edges selected from a corresponding column. This indicates that a node can connect to at least one or more nodes. This is captured by using disjunction. Since connection to multiple nodes is allowed, all other nodes in the same row must be disabled. For example, in Figure 6.5, if $e_{1,1}$ and $e_{3,1}$ are selected, then $e_{1,2}$, $e_{1,3}$, $e_{1,4}$, $e_{3,2}$, $e_{3,3}$ and $e_{3,4}$ are disabled. Thus, this allows a_1 to connect both b_1 and b_3. Meanwhile, b_1 and b_3 cannot be connected by other nodes other than a_1. This is captured by the second sub-formula.

6.3 Quantity of Nodes and Edges

While establishing bounds on the number of instances of given classes allows for a degree of control over the model size, it is sometimes desirable to achieve more fine-grained control. For example, the McCabe cyclomatic complexity metric could be calculated on an instance of a metamodel representing the control-flow graph of a program [McCabe, 1976]. This metric is calculated as $J - K + 2$ for a connected graph with J edges and K nodes. Specifying a desired value, or value range, for this metric simply involves specifying a linear inequality over the size of the graph.

In order to specify the generation of graphs satisfying constraints on the number of nodes and edges, two phases of encodings are needed. Phase 1 decides an appropriate *list of nodes* to be selected by the SMT2 solver, along with the number of edges (J). This is shown in Formula 6.6 (Note that K and J are SMT2 integer constants, their values are not fixed unless users write additional constraints.).

$$(\bigwedge_{i=1}^{b(N)} (K = i) \rightarrow (J \leq \begin{pmatrix} i \\ 2 \end{pmatrix})) \wedge ((\sum_{i=1}^{b(N)} N_i) = K) \wedge (\bigwedge_{i=1}^{b(N)} 0 \leq N_i \leq 1) \tag{6.6}$$

The first sub-formula covers all the possibilities for a value of K can be assigned, and it also restricts the number of edges (J) to be within the possible number of nodes that can appear in an instance. That is J cannot exceed the maximum possible number of edges for a corresponding value of K assigned. The second sub-formula determines which particular nodes get selected from the set of graph nodes V_G. Since we need to know which node(s) to be chosen, we use N_i (an SMT2 integer constant) to encode each graph node from V_G. The last sub-formula constrains N_i to be either 0 or 1, controlling whether it is omitted or selected respectively. Note that the possible values for K varies from 1 and $b(N)$ inclusive. Thus, the results of phase 1 are that an appropriate number of edges J is computed and a list of nodes ($N_i's$) are chosen.

To understand how this works, please consider the following example.
Suppose $b(N) = 3$ and a user specifies 3 ($K = 3$) nodes to be included in the graph. By expanding Formula 6.6, we have the following formulas [1]:

$$
\begin{aligned}
&(K = 1) \rightarrow (J \leq 0) \quad \wedge \\
&(K = 2) \rightarrow (J \leq 1) \quad \wedge \\
&(K = 3) \rightarrow (J \leq 3) \quad \wedge \\
&((N_1 + N_2 + N_3) = K) \quad \wedge \\
&(0 \leq N_1 \leq 1) \wedge (0 \leq N_2 \leq 1) \wedge (0 \leq N_3 \leq 1) \wedge \\
&(K = 3)
\end{aligned}
$$

Observe the formulas above, we note that when a user specifies $K = 3$, this forces each N_i is to be selected. That is each N_i must be assigned with value of 1. $K = 3$ also implies that $J \leq 3$, which means that the number of edges allowed in the graph cannot exceed 3. After checking these formulas with an SMT2 solver, a list of nodes (N_1, N_2, N_3) and the number of edges J can be *decided.* One possible assignment from an SMT2 solver is $N_1 = 1, N_2 = 1, N_3 = 1, K = 3$, and $J = 3$.

In phase 2, since the nodes ($N_i's$) to be presented in the graph are already known (from phase 1), those nodes that are not chosen at phase 1 are switched off and those that are chosen are turned on. Based on the set of nodes and the number of edges that have been chosen, a subset of SMT2 constants (S) that encode edges from the 2D-array which describes each association relationship is specified.

In order to keep our formula simple at this phase, we flatten S into 1D-array.

[1] To help readers understand the formula, we put the complete SMT2 formula at http://www.rise4fun.com/Z3/QkP9 for validation.

We now can sum up all SMT2 constants that encode edges from S. Formula 6.7 uses an SMT2 integer constant S_i which can only be assigned a value of 1 or 0, to represent whether or not an edge is selected. This formula states that the summation of all such SMT2 constants meets the required number of edges J computed from Formula 6.6. Note that Formula 6.7 can also be used for computing the number of edges connected for a specific node in a general case. For example, a user may require a certain number of links to be established in each instance.

$$(\sum_{i=1}^{|S|} S_i) = J \tag{6.7}$$

6.4 Examples: Class Cohesion and McCabe Complexity

This section demonstrates two examples to show the generation of instances having graph properties. The first example is the generation of a particular class cohesion value and a limited length of a call graph, while the second example shows how to generate a graph with a particular McCabe complexity value.

6.4.1 Class Cohesion and Call Graphs

For the first example, we use the same metamodel presented in Figure 6.1, and provide sample bounds for the non-abstract classes in this metamodel as shown in Figure 6.6. This subset of the metamodel represents the relationship (a field can be *accessed* by multiple methods, and a method can *call* multiple methods) between $Class$, $Method$ and $Field$. From this metamodel, instances of programs with a particular value of a cohesion metric (LCOM) can be generated [Chidamber and Kemerer, 1994b], and with a particular depth of the call graph [Li and Henry, 1993].

LCOM is a structural class cohesion metric that measures the number of disjoint components in a graph, where each node represents a method and an edge indicates that two methods share at least one common field. In order to

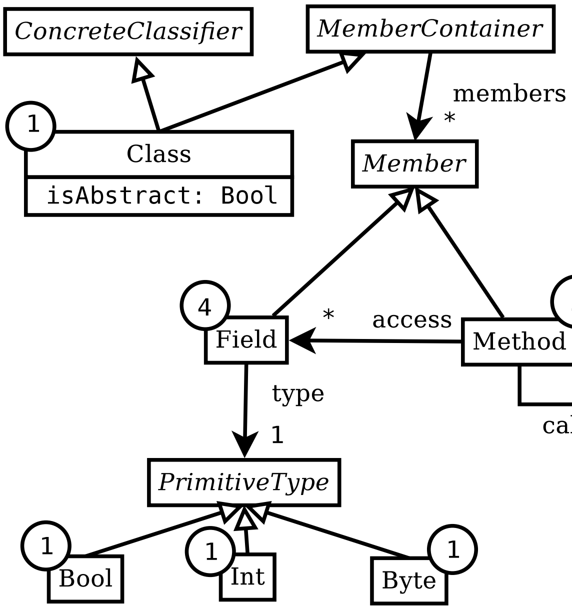


Figure 6.6: A subset of the Java metamodel represented as a bounded graph representing relationships between classes and their members, which are fields or methods.

generate instances, an SMT2 solver is used to compute how the graph should be connected according to the bounds defined over the metamodel.

Thus, an SMT2 solver is used for finding assignments for the constraints encoded in Formula 6.8:

$$(\bigwedge_{k=1}^{c} \bigvee_{j=1}^{b(Method)} m_j = k) \wedge (\bigwedge_{j=1}^{b(Method)} 1 \leq m_j \leq c) \tag{6.8}$$

Here, c is the desired value for the LCOM metric specified by the user, for example LCOM should be evaluated to 3 that means a graph has 3 connected components. Thus, c denotes the number of connected components. We use an SMT2 integer, m_j to encode a method. The possible values of an m_j can get indicates that whether the corresponding method is connecting to another.

If two methods are assigned the same integer, this indicates that they are connected in the graph (one connected component), otherwise they are disconnected. Note that we require that c varies between 1 and $b(Method)$ inclusive, that is because the connection of a list of *methods* varies from not being connected at all to being fully connected.

To understand how Formula 6.8 works, we use the following example. Suppose we have five *methods* (*method*1, *method*2, *method*3, *method*4, and *method*5), and we would like them to form a graph with three connected components. That means three different integer values should be assigned to those *methods* to indicate three different connected components ($c = 3$). Any two *methods* get assigned to the same integer means that they are connected. Now, we expand Formula 6.8, we get the following terms [2]:

$((m_1 = 1) \vee (m_2 = 1) \vee (m_3 = 1) \vee (m_4 = 1) \vee (m_5 = 1)) \quad \wedge$
$((m_1 = 2) \vee (m_2 = 2) \vee (m_3 = 2) \vee (m_4 = 2) \vee (m_5 = 2)) \quad \wedge$
$((m_1 = 3) \vee (m_2 = 3) \vee (m_3 = 3) \vee (m_4 = 3) \vee (m_5 = 3)) \quad \wedge$
$(1 \leq m_1 \leq 3) \wedge (1 \leq m_2 \leq 3) \wedge (1 \leq m_3 \leq 3) \quad \wedge$
$(1 \leq m_4 \leq 3) \wedge (1 \leq m_5 \leq 3)$

In the expanded formulas above each m_j denotes a *method*, for example m_1 denotes *method*1. Now observe these formulas, we note that three of the $m_j's$ can be assigned with value of 1,2 and 3. Because of the additional constraints $1 \leq m_j \leq 3$, this guarantees that the remaining two $m_j's$ share some integers with other $m_j's$, since there are only 3 possible integer values to be assigned to 5 *methods*.

One possible solution (Note that one might get a different solution, this depends on the SMT solver.) derived from Formula 6.8 is shown in Figure 6.7, where $m2$, $m3$ and $m4$ are assigned with the same value of 3, indicating that *method*2, *method*3 and *method*4 are connected together in the graph. That means that these three methods share at least one common field. On the other hand, $m1$ and $m5$ are assigned values that are different from others. This specifies that *method*1 and *method*5 are disjoint from other methods. Therefore, a graph with an $LCOM$ value of 3 has been constructed, and we can enumerate each successful assignment for this formula to get every possible graph with this $LCOM$ value.

Now that how the graph is structured for an $LCOM$ value of 3 is known, sharing and non-sharing formulas on the five methods can be applied. More specifically, the column can be located from the 2D-array capturing the *ac-*

[2] To help readers understand the formula, we put the complete SMT2 formula at: http://www.rise4fun.com/Z3/s1jz for validation.

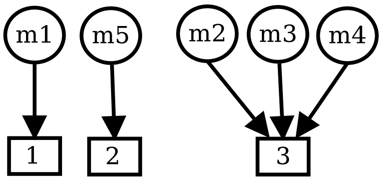


Figure 6.7: One of the successful assignments for the formula for deciding how the graph is connected, based on the bounds defined on metamodel in Figure 6.6 and given a desired $LCOM$ value of 3. Here $m1$ through $m5$ represent methods, and the numbers 1 through 3 represent three sets in a partition of these methods.

cess association, and two lists are constructed, one each for the sharing and non-sharing nodes. In this example, we simply use weak sharing formulas on $method2$, $method3$ and $method4$, and strong non-sharing formulas on $method1$ and $method5$. As Figure 6.8 shows, $method2$, $method3$ and $method4$ all have access to $field3$, while $method1$ and $method5$ have accesses to $field2$ and $field4$ respectively.

To generate the call graph with a depth of 3 for $Method$ in the metamodel in Figure 6.6, Formula 6.1 is applied to the association $calls$. Figure 6.8 shows the series of method calls giving a depth of 3: $method1$ calls $method4$ and $method4$ calls $method5$ which calls $method3$.

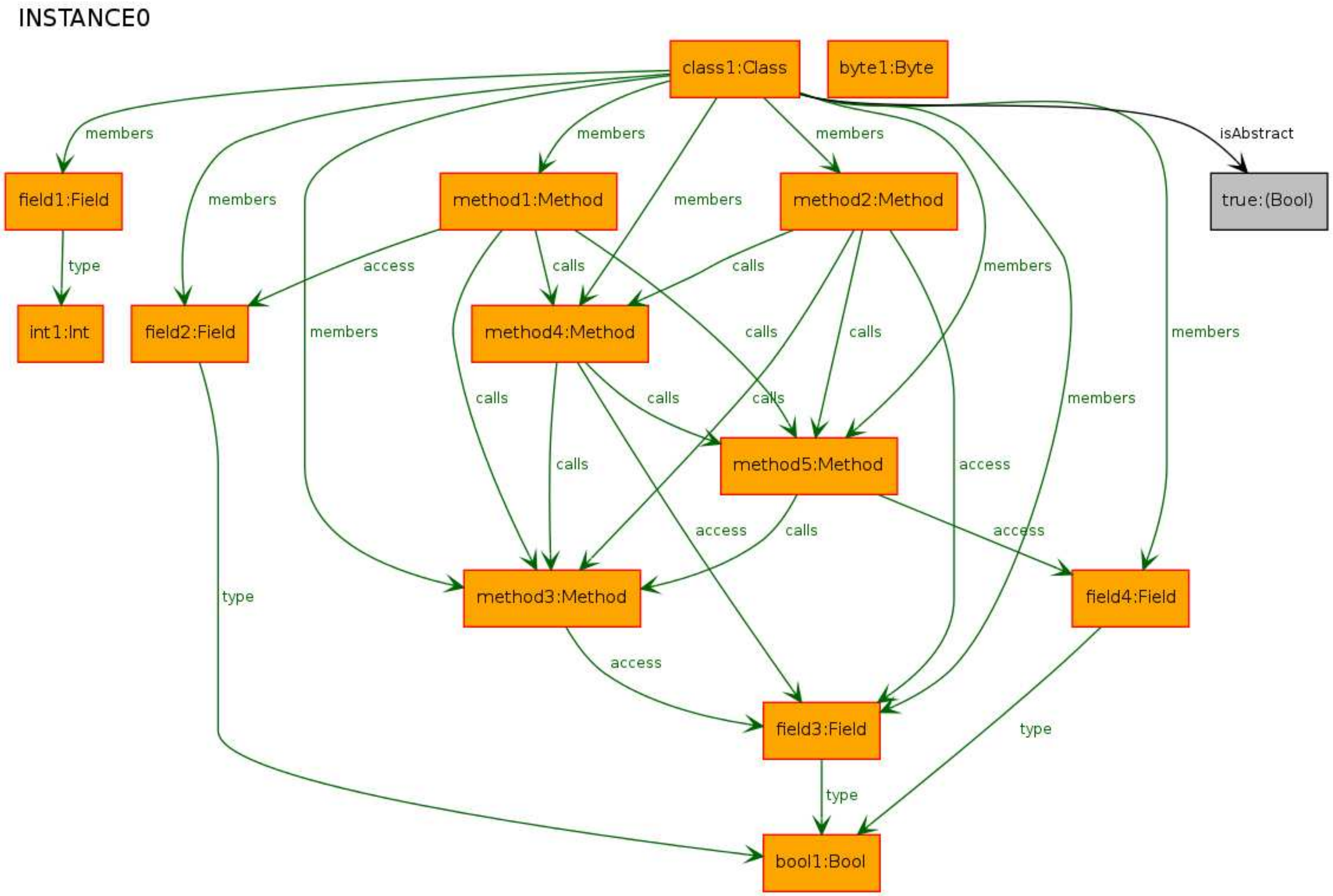


Figure 6.8: A generated instance of the Java metamodel from Figure 6.6. This Java program has an $LCOM$ value of 3, as well as a call-graph depth of 3.

6.4.2 McCabe Complexity

McCabe complexity (cyclomatic complexity) was first introduced in 1976, it measures the number of independent paths of a program's source code so that developers are sensitive to the number of test cases needed for testing a program [McCabe, 1976]. The program source code is represented as a control flow graph, and a control flow graph is a directed graph. Thus, in terms of this representation, McCabe complexity is a metric that measures the complexity of a graph. The formal measurement of McCabe complexity M can be calculated as the following formula (For simplicity reason, we consider a graph that only has one connected component.).

$$\begin{gathered} M = J - K + 2 \\ where \\ J \text{ specifies the total number of edges in a graph.} \\ K \text{ specifies the total number of nodes in a graph.} \end{gathered} \tag{6.9}$$

For example, Figure 6.9(b) has 7 nodes, 8 edges and 1 connected component in total within a graph. Thus, the McCabe complexity of this graph is calculated as $8 - 7 + 2$ which is 3.

In section 6.3 a way of generating a particular number of nodes and edges in graph is provided, and using this we can produce a graph that has a particular McCabe complexity of M. In order to do so, for a given McCabe complexity of M, the number of edges (J) and nodes (K) in a graph is calculated. This calculation is done by using an SMT2 solver to solve the conjoined formulas 6.6 and 6.9 with respect to the bounds defined on the class in a metamodel. Based on the results of this calculation, the exact number of edges (J) and nodes (K) that make equation 6.9 evaluate M is known. Now a graph with McCabe complexity of M can be generated by using Formula 6.7 to get a correct number of subsets of edges that equals J.

For example, suppose we wanted to generate a graph that has a McCabe complexity of 3, with respect to the bound of 8 defined on *Node* class in the metamodel in Figure 6.9(a). First, the correct number of edges (J) and nodes (K) are calculated by using an SMT2 solver to solve the conjunction of $3 = J - K + 2$

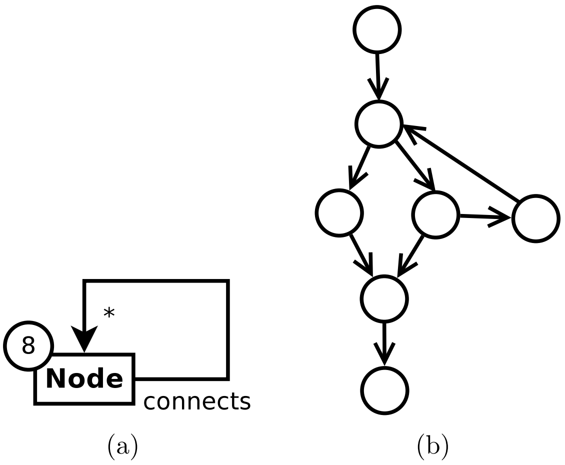


Figure 6.9: A metamodel that describes directed graphs with a bound of 8 on the *Node* class in Figure 6.9(a). An instance, with a McCabe complexity of 3, of the metamodel in Figure 6.9(b).

and formula 6.6 with respect to the bound of 8 for *Node* class. One of the possible solutions found by SMT2 solver for J and K is 7 and 6. This means a graph has to contain 7 edges and 6 nodes in total. The assignment of the formula 6.6 also tells that 6 nodes have been selected from the universe. Thus, the formula 6.7 now can be applied to generate graphs that has a McCabe complexity of 3. Figure 6.10 shows one of the instances of metamodel in Figure 6.9(a) that has a McCabe complexity of 3.

6.5 Evaluation

6.5.1 Implementation

We have extended our ASMIG tool to support graph-based criteria instance generation by adding an extra component as shown in Figure 1.4. This component collects information about the association in a metamodel during the translation stage, and translates the scenarios specified by a user to a set of SMT2 formulas. Each satisfied assignment for the formulas is then interpreted as an instance having particular graph properties. We have evaluated ASMIG on a machine with a CPU of Intel Core 2 Duo (E7500) 2.93GHz, and a 4GB memory, and the results

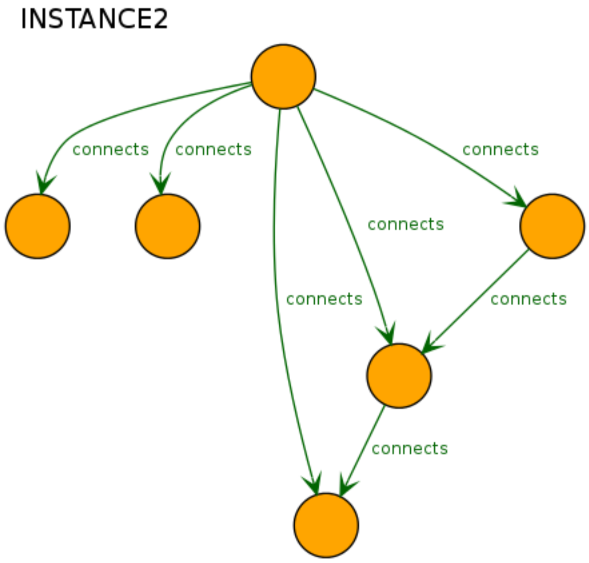


Figure 6.10: A generated instance (6 nodes and 7 edges) of the metamodel in Figure 6.9(a) that has a McCabe complexity of 3.

are shown in Table 6.1.

6.5.2 Results

Since graph-based criteria focus on a particular association in a metamodel, Chidamber and Kemerer (CK) metrics are used to evaluate this approach against a subset of the Java programming language metamodel [Chidamber and Kemerer, 1994a]. This metamodel is depicted in Figure 6.1. We did not choose the previous metamodels as a test suite because the formulas described in this chapter focus on a specific association within a language metamodel. It is sufficient enough to test these formulas on a specific association in a metamodel because a metamodel that describes either a general purpose of programming language or domain specific language has the same language properties such as method invocation, class inheritance, attribute references, etc. We are particularly interested in CK metrics because not only were they proposed for measuring different aspects of an object-oriented design but also they can be treated as certain graph properties involving numeric constraints. CK metrics has a total of 6 different metrics [Chidamber and Kemerer, 1994a]:

1. **Weighted methods per class(WMC):** WMC is calculated by summing the complexities of all methods in a class. However, the authors of WMC have not given a precise definition of complexities in order to allow for the most general application of this metric. In our view, we consider the complexities as the count of the methods in a class.

2. **Depth of inheritance (DIT)**: DIT of a class is calculated based on the maximum length from the root of the inheritance tree to that class.

3. **Number of children (NOC)**: NOC is calculated by counting the number of immediate subclasses that a class can have.

4. **Lack of cohesion in methods (LCOM)**: LCOM is calculated by the number of non-intersecting sets of methods based on the common usage of instance variables.

5. **Coupling between object classes (CBO):** CBO is calculated by counting the number of non-inheritance related couples with other classes. Two classes are considered coupled classes if the method of one class reference at least one attribute or invoke at least one method from the other class.

6. **Response for a class (RFC):** RFC is calculated by the size of response set of a class, and the response set is defined as the set of distinct methods can be invoked from that class.

As can be seen, these 6 metrics involve measuring quite a number of different aspects of object-oriented design based on numeric values. Thus, we consider the set of CK metrics as a suitable test suite for testing instance generation involving both graph and numeric constraints. For each metric, we apply different formula described in this chapter. For example, for metrics WMC and NOC, we apply Formula 6.7 to a specific association because Formula 6.7 precisely constrain the number of edges to be selected, and both WMC and NOC require a specific number of links associated with a specific class. For DIT metric, it is easy to see that we can just apply Formula 6.1 to a specific reflexive association. For LCOM metric, the example in section 6.4.1 have already shown the detailed steps.

We have tested ASMIG against different bounds for each metric to measure its scalability. The graph in Figure 6.11 shows that the translation time is affected by the bounds defined. In general, the translation and average finding time is proportional to the size of bounds allocated on both association-ends. However, ASMIG tries to utilise cache mechanism as much as possible to prevent formula regeneration, and this depends on a specific association defined in a metamodel. Table 6.1 provide more detailed results and the bounds chosen for testing ASMIG. The "Total Bound" in Table 6.1 specifies summation of the individual bound allocated for two association ends, and all bounds are manually allocated. The specific bound allocated for each specific class can be found at our website [3]. In reality, it is rare that a large bound would be allocated to two particular association ends. Therefore, a maximum bound of 10 is chosen for both association ends. The average finding time in Table 6.1 is calculated based on the generation of 100 instances for each metric. Figure 6.11 also reflects that the translation time significantly increases when we set bound equal to 10, but it is still considered to be fast (below 500 milliseconds). Thus, this indicates that even under a very rare situation the translation time ASMIG takes is still very acceptable.

Interestingly, ASMIG cannot generate two coupling metrics: Coupling between object classes (CBO) and Response for a class (RFC). The reason that ASMIG cannot work for metrics CBO and RFC is that Formula 6.1 to Formula 6.7 only work on an individual association. In other words, these formulas work with a single 2D-array. However, both CBO and RFC require additional constraints over more than one single array. For example, in order to constrain CBO, the constraints must be defined over at least two associations: one for attribute and one for method invocation. In the meantime, an intermediate constraint is also needed in order to specify the constraints for those two associations actually referring to the same class. Similarly, for RFC, we also need to express a constraint over two associations: one for calculating number of distinct methods, and one for calculating the size of response set of a class. An intermediate is also needed here to express that the number of response set of a class depends on the number of distinct methods of that class. However, the formulas listed in this chapter do not capture such constraints over two arrays. Thus, this type of

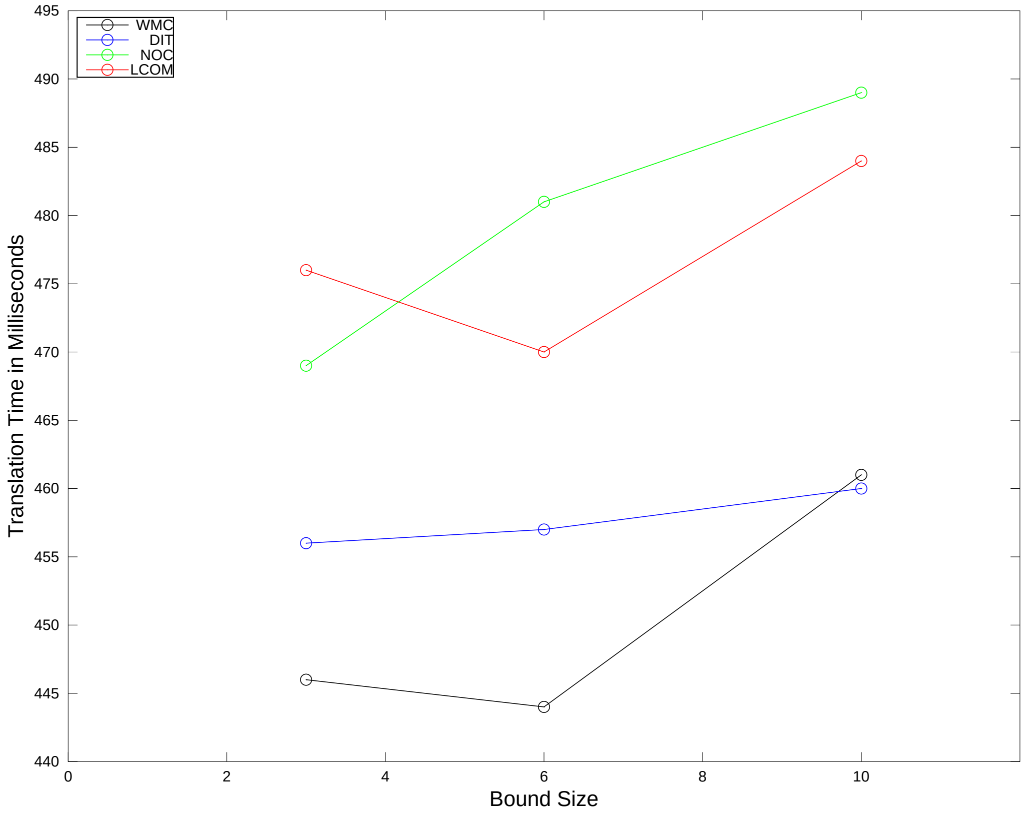


Figure 6.11: Translation time against different size of bound for four metrics. Each point in this graph represents one specific translation time that ASMIG takes based on a specific bound. Points and bounds are derived from Table 6.1.

Chidamber and Kemerer Metric	Metric Value	Total Bound	Time in ms	
			Translate	Avg Find
Weighted methods per class (WMC)	2	3	446ms	31ms
	3	6	444ms	59ms
	5	10	461ms	120ms
Depth of inheritance (DIT)	2	3	456ms	53ms
	4	6	457ms	54ms
	8	10	460ms	180ms
Number of children (NOC)	2	3	469ms	64ms
	4	6	481ms	65ms
	8	10	489ms	180ms
Lack of cohesion in methods (LCOM)	2	3	476ms	63ms
	2	6	470ms	42ms
	3	10	484ms	138ms

Table 6.1: Results of generating 100 instances of metamodel in Figure 6.1. Each metric was constrained using three values, and the calculated bounds are shown, as well as two measures of the time taken to generate appropriate instances.

constraints is not supported by the current version of ASMIG.

6.6 Summary

This chapter describes a *unique* technique that provides a set of translation rules which can be used to encode some scenarios based on common graph properties. With this technique, the instances generated from a metamodel go beyond making a contribution to coverage criteria, as they can be used for other purposes such as testing a compiler, refactoring tools, metrics calculator, etc. The examples and evaluation results shown in this chapter demonstrate feasibility for such purpose. However, the disadvantage of this technique is that the current version of ASMIG cannot capture a scenario that requires constraints over two or more associations. This limits instance generation to a certain number of graph properties, but we still believe this technique uniquely provides a degree of instance generation for achieving graph properties.

[3] http://www.cs.nuim.ie/~haowu/ASMIG/Results/GraphBased

Chapter 7

Conclusion

The high abstraction level provided by metamodels makes the metamodeling approach popular among software engineers. However, one of the biggest drawbacks is that it does not naturally provide a way of generating metamodel instances. The instances are particularly important for software engineers to test their metamodels. However, generating instances from a metamodel is not an easy task. The challenge here is that the generated instances have to satisfy both structural and OCL constraints defined over a metamodel. Furthermore, software engineers may also seek more meaningful instances that are beyond satisfying structural and OCL constraints such as achieving coverage criteria, testing a compiler or metrics calculator.

The contribution of the research presented in this thesis is that it presents a new way consisting of two approaches, that uniquely combine graph representation and Satisfiability Modulo Theories (SMT) to the problem of metamodel instance generation. The first approach presents a new way of generating metamodel instances by translating a metamodel to an SMT problem via a *bounded graph* representation (Chapter 4). The second approach investigates generating meaningful metamodel instances based on two techniques. The first technique generates instances that meet *partition-based coverage criteria* by using criteria formulas to further constrain the entire generation process (Chapter 5). The second technique aims to generate instances that satisfy *graph-properties based criteria* by encoding graph properties into SMT2 formulas according to different scenarios (Chapter 6). The two approaches have been prototyped into our tool

ASMIG, to demonstrate the feasibility of automatic metamodel instance generation.

This chapter concludes the thesis with a summary of the key features behind our approaches, a discussion of our approach's capability compared to others, and an outline of direction for future work.

7.1 Discussion

The key feature of this research is a new foundation that is based on an intermediate representation (Bounded Attribute Typed Graph with Inheritance) that naturally captures the definition of a metamodel. Two techniques governing instance generation, achieving partition-based coverage criteria, and graph-based criteria, have been successfully implemented in our automated tool, ASMIG. We first list these features in Table 7.1, and then assess our solution against the other main approaches. We then use a more detailed Table 7.2 to compare our solution against the other main approaches, in order to identify the relative advantages and limitations of our solution.

We introduce a new graph concept: a Bounded Attributed Type Graph with Inheritance to naturally support the metamodeling approach and this concept acts as an intermediate representation to facilitate SMT-based instance generation. Graph grammars also use attribute type graphs with inheritance. However, without bounds on the graph, the program for generating metamodel instances may never terminate. We differ from graph grammars (described in section 2.3.1.3) by adding an extra bound for each typed graph node to bound our search space, and this guarantees the *termination* of our generation process. We consider a metamodel structure consisting of classes, attributes (Integer, Boolean and Enumeration types), inheritance and the most frequently used kinds of association (uni/bi-directional). However, not all of these metamodel structural features are supported by other approaches. For example, approaches like Echo fails to support multiple inheritance relationships (this is shown in section 4.5.3). This is because Echo is based on Alloy's specification which does not allow multiple inheritance relationships. Other approaches like the SMT bit-vector does

Approach	Bounded ATGI	Metamodel Structure	Object Constraint Language	Partition-based Coverage Criteria	Graph-based Criteria	Instance Enumeration	Reasoning Engine	Tool Support
Our Approaches	■	■	⊔	■	⬓	■	SMT2 Solver	■
Alloy	□	⬓	⬓	□	⊔	■	SAT Solver	■
Constraint Programming (CP)	□	■	⊔	□	□	⊔	ECL ^{i}PS^e	■
Graph Grammars	⬓	■	⊔	□	□	□	AGG	□
Echo	□	⬓	⬓	□	□	■	Alloy	■
USE	□	■	⬓	□	□	■	Alloy	■
SMT bit-vectors	□	⬓	⬓	□	□	□	SMT Solver	□
Description Logic	□	■	□	□	□	⊔	CSP	■

Table 7.1: A comparison of key features between our approach and other approaches. A solid box indicates that a feature is fully supported, a half of a solid box indicates that a feature is partially supported, a mostly empty box indicates that a relatively small portion of a feature is supported, and an empty box indicates that a feature cannot be supported at all.

not support general metamodel structural features such as association. This is because it is only concerned with OCL data types [Soeken et al., 2010, 2011b].

We currently support a relatively small portion of OCL features, namely constraints on attributes, quantifiers over objects and navigation on an association. This puts a limitation on our current approaches for those metamodels that frequently use advanced OCL features such as operations over a collection data type. The reason for this limited support of OCL is that our approaches lack a formal language for describing constraints on nodes and edges translated from a metamodel. This is because the graph representation used during our translation does not provide language features for capturing OCL constraints. Though it is possible to use OCL to write graph-based constraints, OCL is more than a declarative language. For example, features involving expressions like conditions and loops make the translation very difficult and expensive for describing a simple graph property such as a directed acyclic graph. Other approaches like SMT bit-vector, Constraint Programming (CP), Alloy [Anastasakis et al., 2007; Kuhlmann et al., 2011; Soeken et al., 2011b,b], support quite a number of OCL features such as collection data types (sequence, bag and set), and operations (union, intersect) on those data types. Most of them directly benefit from the first-order relational algebra provided by Alloy. Thus, a substantial subset of OCL language features can be easily mapped to this algebra [Anastasakis et al., 2007; Bordbar and Anastasakis, 2005; Kuhlmann et al., 2011].

However, the research built around Alloy suffers from the same fundamental problem as Alloy does. Alloy translates a relational specification to SAT. SAT-solvers are designed to solve boolean satisfiability problems. Any constraints that involve integer arithmetic operations causes bit-blasting on Alloy's engine (kodkod) [Torlak, 2009]. On the other hand, SMT solvers have a list of dedicated algorithms which perform operations on integers or real numbers. The translation from a particular integer equation, especially those with inequalities, to SMT-instances is straightforward, since SMT-instances have a more powerful expressiveness than the plain SAT-instances. Using SMT solvers as back-end reasoning engines are popular in many areas [Armando et al., 2009; Cordeiro et al., 2009; Milicevic and Kugler, 2011; Tianhai Liu, 2012]. Though constraint programming approach is also popular, it requires much more expertise and programming skills for a particular problem than SAT/SMT [Puget, 2004]. Furthermore, SMT solvers are more promising in terms of future research directions [De Moura and Bjørner, 2011; Malik and Zhang, 2009].

The solution described in this thesis provides a technique that uniquely generates metamodel instances based on partition-based coverage criteria. One can utilise Alloy's symmetry detection algorithm to break symmetries in the instances to achieve a certain degree of coverage.[Crawford et al., 1996a; Shlyakhter, 2007; Torlak and Jackson, 2007]. However, the purpose of using a symmetry breaking algorithm is to remove similarities between graphs, not for partition-based instance generation [Crawford et al., 1996b; Katebi et al., 2012; McKay, 1981].

Thus, the algorithm produces irrelevant instances in terms of the coverage criteria.

We also provide a unique technique for generating instances that meet certain graph-based properties, and these properties are particularly important to produce instances for other purposes. For example, users may require a metrics oriented instance generation like in the McCabe complexity example we illustrated in section 6.4.2. This unique technique enables us to provide a degree of instance generation for meeting graph properties unlike other approaches. For example, graph grammars deal with graphs, but they do not support any graph-properties based instance generation [Ehrig et al., 2009]. In order to support this instance generation, a set of new grammar rules has to be created. Alloy's first-order relation language can be used to describe some graph properties such as a directed acyclic graph. However, it performs poorly when the graph properties involve numeric constraints such as depth of inheritance tree, lack of cohesion between two classes.

Approaches like CP, graph grammars, SMT-bit vectors and description logics verify a metamodel by constructing only one single instance. Therefore, these approaches have a major limitation when producing a set of instances (e.g., a test suite) from a metamodel [Cabot et al., 2007; Cadoli et al., 2004; Ehrig et al., 2009; González Pérez et al., 2012; Macedo and Cunha, 2013]. Using logic formulas to encode the problem is elegant and deterministic since instance enumeration can easily be achieved by adding extra formulas to prevent all previous assignments found by SAT/SMT solvers [Torlak, 2009; Wu et al., 2013].

In terms of implementation, most of existing approaches provide fully automatic tool support except for the approach using SMT bit-vector theory [Soeken et al., 2011b]. However, all tools (except for Alloy) are built primarily for the purpose of generating UML or metamodel instances. By comparison, ASMIG is designed to not only provide metamodel instance generation but also to tackle general model finding. ASMIG is a relatively small, but fully automatic tool, and it consists of about 23,000 lines of source code with around 7,000 lines dedicated to the core engine. Its core engine can be plugged into other tools for solving general constraint problems such as n-queens, the Sudoku example in section 3.8. The downside is that it requires knowledge of using APIs from the core engine to

	Metamodel				OCL			Coverage		Graph	Tool
Related Approaches	Attribute	Association	Multi-inheritance	Supported Size	Navigation	Collection	Quantifier	Attribute	Association	Metrics	Automation
ASMIG	✓	✓	✓	*L*	✓	✗	✓	✓	✓	✻	✓
Alloy	✓	✓	✗	*M*	✓	✓	✓	✗	✗	✗	✓
Constraint Programming	✓	✓	✓	*M*	✓	✗	✓	✗	✗	✗	✓
Graph Grammars	✓	✓	✓	*S*	✓	✗	✓	✗	✗	✗	✗
Echo	✓	✓	✗	*M*	✓	✗	✓	✗	✗	✗	✓
USE	✓	✓	✓	*M*	✓	✗	✓	✗	✗	✗	✓
SMT bit-vectors	✓	✗	✗	*S*	✓	✓	✓	✗	✗	✗	✗
Description Logic	✓	✓	✓	*S*	✗	✗	✗	✗	✗	✗	✗

Table 7.2: A detailed comparison with other approaches. NOTE: a ✓ means a feature can be fully supported, and a ✻ means a feature is partially supported and a ✗ means that a feature cannot be supported. An *L* denotes large size metamodels, an *M* denotes medium size metamodels and an *S* denotes small size metamodels.

encode the constraints before applying the tool.

To be able to identify both the advantages and disadvantages of ASMIG, we provide Table 7.2 to illustrate a detailed comparison with other approaches. Table 7.2 lists detailed features for five main categories that are: *Metamodel*, *OCL*, *Coverage*, *Graph* and *Tool*. The first two categories focus on metamodel structural and OCL constraints, the next two focus on the unique features provided by ASMIG, and the last one shows the automation of each tool. To the best of our knowledge, we are the first to present the comparison of related work at different levels: metamodel, OCL, coverage, graph and tool.

For metamodel, ASMIG has an advantage over Alloy-based approaches like Echo. This is because Alloy's specification cannot support features like multiple inheritance relationships. ASMIG uses the concept of Bounded Attributed Type

Graph with Inheritance that can naturally capture all metamodel features such as associations and multiple inheritance relationships. Apart from that, ASMIG is also capable of handling quite a wide range of different size metamodels compared to other approaches, as the results have shown in section 4.5.3.

Compared to SAT-based approaches, the disadvantage of ASMIG is that we support less OCL features. In general, SMT instances are more expressive than pure SAT instances. This is because SMT solvers provide different kinds of theories. In fact, SMT-based approaches like SMT-bit vector shows that it is possible to use SMT bit vector theory to encode advanced OCL data types such as sets and bags [Soeken et al., 2011b]. However, these encodings must be manually provided by users. This is impossible for users who are not familiar with SMT encodings. In terms of our approach, the disadvantage here is that the intermediate representation (Bounded Attribute Typed Graph with Inheritance) used during our translation lacks a way of integrating OCL advanced language features into the graph. This is because OCL is not designed for expressing graph structural elements and constraints over a graph. The advantage is that using this graph representation we can naturally capture metamodel structural features. Thus, the trade off for our approach here is between graph structural elements and OCL language advanced features. However, we still consider ASMIG to be a worthy tool in the area, as this can be seen from both Table 7.2 and the comparison in section 4.5.3.

ASMIG provides two unique techniques: partition-based and graph-properties based instance generation. These two features enable users to generate metamodel instances more closely for testing purposes. With partition-based instance generation, a user is able to generate metamodel instances according to predefined partitions, and this allows users to perform tests on their metamodels using partition-based criteria. With graph-properties based instance generation, users are able to generate metamodel instances that concern certain graph properties. For example, two nodes cannot share other nodes. Though there are graph properties that may not be captured by this technique, compared to other approaches, we consider this technique as a unique feature for metrics oriented instance generation as demonstrated in section 6.5.

7.2 Future work

The research presented in this dissertation extends the knowledge of solving the metamodel instance generation problem to another dimension: a combination of graph and Satisfiability Modulo Theories (SMT). Though our approaches build a new foundation that naturally supports generating metamodel instances, a number of problems still remain at both the graph (language) and the SMT (formula) level. We discuss these issues in the sections that follow.

7.2.1 Shared SMT2 Formulas

Though an SMT solver can generally solve a large number of formulas within a short amount of time, it still would be a valuable feature if one could remove any redundant formulas to achieve faster solving. The translation scheme in our approach does not remove redundant formulas and this leads to a waste of memory and time spent on translation. For example, translation step 6 in the graph colouring example, in Figure 4.11, causes redundant formulas. Thus, one future direction is to add a new mechanism that can detect and share common SMT2 formulas at the translation stage, and remove redundant formulas for efficiency. One possible solution is that this new mechanism could use a hash function to detect possible shared SMT2 formulas, and build reduced abstract syntax trees for these formulas.

7.2.2 Unsat Core Analysis

A formula encoding a problem found to be unsatisfiable (unsat) means that at least one contradiction exists in the formula, and extracting such a contradiction is useful for further analysis on the cause of conflict in the encodings. Most of the SMT2 solvers support a feature called *unsat core extraction*, where this feature allows users to retrieve a set of formulas that trigger an unsatisfiable result, when an SMT solver fails to find an assignment for the given formulas. Thus, it will be useful to have a feature that can analyse unsat formulas and reflect them in the problem domain. For example, identifying conflicting OCL constraints defined on a metamodel. This work could be extended by adding an extra component that can track every formula (including sub-formulas) in the translation engine. An analysis engine could identify the formulas from SMT unsat core, retest the entire formula by removing part of the formulas until a successful assignment is found, and interpret the formulas back to the metamodel and OCL constraints.

7.2.3 Extending OCL Support

The need for supporting a large subset of OCL constraints is necessary for those metamodels which rely heavily on OCL. In our current approach, only a small subset of OCL constraints are supported. Thus, it would be useful to extend our current approach to support other OCL features: for example, operations on different data types such as strings or collections [Büttner and Cabot, 2012]. The difficulty here is that a pre-defined size for collection data types is needed before encoding the actual contents into logic formulas. Existing work includes an encoding that captures data collection types by using SMT bit-vector theory [Soeken et al., 2011b]. It would be valuable to first incorporate this work into our approach, and then extend it to operations on data collection types such as *select* and *reject*. A technique could then be proposed for calculating the pre-defined size for a collection by scanning through any operators which involve defining the size of that collection. This technique would thus require a detailed analysis of every OCL expression involving numeric calculations.

7.2.4 Special Graph-Properties Based Language

Though a technique that can handle some graph properties has been discussed in this thesis, this technique does not scale well enough to support more general graph-properties such as a bipartite graph. The main reason is the lack of a formal language that can effectively capture those graph-properties at a more general level. Thus, a promising future direction is to design a new graph-properties based language that can easily capture more complex graph structures than a metamodel can have. This would lead to a new translation scheme that could efficiently encode these properties into SMT2 formulas. With this graph-properties based language, metamodel and OCL constraints for describing graph properties can be transformed into that language. However, the idea of having this language is to capture more complex graph structures than metamodel can possess. This is because solving complex graph constraints is much more general topic than the focus of this particular research (metamdoel instance generation). The challenge in that case would be to allow users to write logical constraints over a graph structure in a simple way without losing the power of expressiveness. This would require a sophisticated design at both the language and formula level. The research presented in this thesis can be viewed as a starting point for combining graph representations (language) with SMT solvers (formula) that with further development would offer an alternative perspective to those tackling more general complex graph constraint problems.

Appendix A

To see why the Sudoku problem presented in section 3.4 can be translated into an SMT problem in polynomial time, we show each formula in Figure 3.9 can be implemented into an algorithm that runs in polynomial time to output final SMT formula. In particular, we consider that each algorithm is measured by its worst-case time complexity [Cormen et al., 2001; Sipser, 1997]. For each of the algorithms listed below, we assume each increment, assignment and comparison statement takes only one step to evaluate. *Expr* in each algorithm denotes a boolean logic formula. $|row|$ and $|column|$ denote the size of a Sudoku grid (number of rows and columns). We use S to denote the total number of steps for an algorithm runs through under worst-case time complexity. Note that in each following algorithm, the operations $\wedge$, $\leq$ and $\neq$ do not get evaluated during the algorithm execution and they are only evaluated by the SMT2 solver.

Proof. For the first formula:

$$\bigwedge_{i=1}^{|row|} \bigwedge_{j=1}^{|column|} 1 \leq C_{i,j} \leq 9 \tag{1}$$

One can implement it into the Algorithm 1:

In Algorithm 1, the variable i in the outer loop gets evaluated for $|row| + 1$ times (one extra evaluation of i for quitting the loop). Similarly, the variable j in the inner loop gets incremented from 1 to $|column| + 1$ every time the outer loop executes. The statement in the inner loop consists of an assignment statement

Algorithm 1 Expand Formula 1

```
Input: A Sudoku Grid
Output: $True \wedge (1 \leq C_{1,1} \leq 9) \wedge (1 \leq C_{1,2} \leq 9) \wedge ... \wedge (1 \leq C_{|row|,|column|} \leq 9)$
Require: $Expr = True \wedge |row| \geq 1 \wedge |column| \geq 1$
for $i = 1$ to $|row|$ do
    for $j = 1$ to $|column|$ do
        $Expr = (Expr) \wedge (1 \leq C_{i,j} \leq 9)$
    end for
end for
```

that gets evaluated 1 time. Every time the outer loop is executed, the inner loop is executed for $|column|$ times. Thus, to calculate total number steps (S) we just add them together:

$$
\begin{aligned}
S &= |row| + 1 + |row|(|column| + 1 + |column|) \\
&= |row| + 1 + |row||column| + |row| + |column||row| \\
&= |row||column| + |column||row| + 2|row| + 1 \\
&= 2|row||column| + 2|row| + 1
\end{aligned}
$$

We use a constant n to denote the Sudoku grid size is n by n. Now we have $S = 2n^2 + 2n + 1$, and by using the big O notation we can write $O(n^2)$ [Cormen et al., 2001; Sipser, 1997]. Note that Algorithm 1 is for a standard 9x9 Sudoku puzzle, if the Sudoku grid size is n, one might change $(1 \leq C_{i,j} \leq 9)$ to $(1 \leq C_{i,j} \leq n)$. ■

Proof. For the second formula:

$$\bigwedge_{i=1}^{|row|} \bigwedge_{j=1}^{|column|-1} \bigwedge_{k=j+1}^{|column|} C_{i,j} \neq C_{i,k} \tag{2}$$

One can use the following algorithm to implement this formula:

Algorithm 2 Expand Formula 2

```
Input: A Sudoku Grid
Output: $True \wedge (C_{1,1} \neq C_{1,2}) \wedge ... \wedge (C_{|row|,|column|-1} \neq C_{|row|,|column|})$
Require: $Expr = True \wedge |row| \geq 1 \wedge |column| \geq 1$
1: for $i = 1$ to $|row|$ do
2:    for $j = 1$ to $|column| - 1$ do
3:       for $k = j + 1$ to $|column|$ do
4:          $Expr = (Expr) \wedge (C_{i,j} \neq C_{i,k})$
5:       end for
6:    end for
7: end for
```

The variable i in the most outer loop gets evaluated for $|row| + 1$ times, and the variable j in the middle loop gets evaluated for $|column|$ times (one extra evaluation of j for quitting the loop). Observe the most inner loop, we found k starts from $j + 1$ up to $|column|$. This means that when $j = 1$, k starts from 2 to $|column|$. This tells us that k gets evaluated for $|column|$ times (one extra evaluation of k for quitting the loop). When $j = 2$, k gets evaluated for $|column| - 1$ times. So the pattern becomes obvious, when j starts from 1 to $|column|$, k gets evaluated for $|column| + (|column| - 1) + (|column| - 2) + (|column| - 3) + ... + 1$ times. That is $\frac{|column|(|column|+1)}{2}$. By the same observation, the assignment statement (line 4 in Algorithm 2) in the most inner loop gets executed for $(|column| - 1) + (|column| - 2) + (|column| - 3) + ... + 1$ times. That is $\frac{|column|(|column|-1)}{2}$. The assignment in step 4 takes one step to evaluate. Now we can compute S (total number of steps):

$$\begin{aligned} S &= |row| + 1 + |row|((|column| + \frac{|column|(|column|-1)}{2} + \frac{|column|(|column|+1)}{2})) \\ &= |row| + 1 + |row|(|column||column| + |column|) \\ &= |row||column||column| + |row||column| + |row| + 1 \end{aligned}$$

We use constant n to denote the Sudoku size is n by n. So $S = n^3 + n^2 + n + 1$, by using the big O notation we have $\mathrm{O}(n^3)$. ■

Proof. For the third formula:

$$\bigwedge_{i=1}^{|column|} \bigwedge_{j=1}^{|row|-1} \bigwedge_{k=j+1}^{|row|} C_{j,i} \neq C_{k,i} \tag{3}$$

One can implement a similar Algorithm 3 as the one for Formula 2.

Algorithm 3 Expand Formula 3

```
Input: A Sudoku Grid
Output: $True \wedge (C_{1,1} \neq C_{2,1}) \wedge ... \wedge (C_{|row|-1,|column|} \neq C_{|row|,|column|})$
Require: $Expr = True \wedge |row| \geq 1 \wedge |column| \geq 1$
for $i = 1$ to $|row|$ do
    for $j = 1$ to $|column| - 1$ do
        for $k = j + 1$ to $|column|$ do
            $Expr = (Expr) \wedge (C_{j,i} \neq C_{k,i})$
        end for
    end for
end for
```

In a similar way to what we have already shown for Formula 2, we can derive the result here. The two formulas (Formula 2 and Formula 3) are identical except that Formula 2 is for specifying rows and Formula 3 here is for specifying columns.

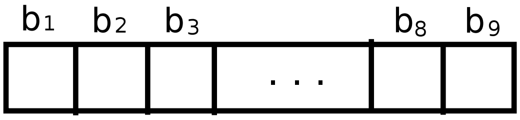


Figure 1: For a standard 9x9 Sudoku puzzle, each two dimensional block is flatten into a list, b_1 denotes the first cell in each block and b_9 denotes the last cell.

As can be seen in Formula 3, the three conjunctions correspond to three loops (nested). Thus, using the same approach for Algorithm and big O notation we have the same time complexity as Algorithm 2, that is $\mathrm{O}\big(n^3\big)$. ■

Proof. The fourth formula describes the rules for each block in the Sudoku puzzle:

$$\bigwedge_{i=1}^{|block_{row}|} \bigwedge_{j=1}^{|block_{column}|} C_{i,j} \neq C_{k,l}, \tag{4}$$

where $i \neq k$ and $j \neq l$, k from 1 to $|block_{row}|$ and l from 1 to $|block_{column}|$. NOTE: when $i = 1$ and $j = 1$ means that the first cell in the block.

This formula specifies that each cell in each block does not share values that the other cells can have. In other words, it indicates that every cell in the block has a unique value. To implement Formula 4 into an algorithm, one can simplify each x by x two dimensional block into a single list with the length of x^2 (denoted as *list.length*). Figure 1 shows this idea and each b_i represents a particular cell from the original block. Now, saying that every cell in the block is unique is the same as saying each b_i in the list is not identical to other $b_i's$ (except for itself). Thus, one can implement this using the Algorithm 4:

In Algorithm 4, for simplicity reason we use notation b_i to denote a corresponding cell in each block. We now analyse the Algorithm 4, we can see that variable i gets evaluated for *list.legnth* times. From $i = 1$ to $list.length - 1$,

Algorithm 4 Expand Formula 4

```
Input: A simplified block represented as a single list
Output: $True \wedge (b_1 \neq b_2) \wedge ... \wedge (b_{list.length-1} \neq b_{list.length})$
Require: $Expr = True \wedge list.length \geq 1$
for $i = 1$ to $list.length - 1$ do
    for $j = i + 1$ to $list.length$ do
        $Expr = (Expr) \wedge (b_i \neq b_j)$
    end for
end for
```

j gets evaluated for $list.length + (list.length - 1) + (list.length - 2) + ... + 1$ times. That is $\frac{list.length(list.length+1)}{2}$. Similarly, the assignment statement in the inner loop gets evaluated for $(list.length - 1) + (list.length - 2) + ... + 1$ times. That is $\frac{list.length(list.length-1)}{2}$. Now, we can add them all together:

$$\begin{aligned} S &= list.length + \frac{list.length(list.length+1)}{2} + \frac{list.length(list.length-1)}{2} \\ &= list.length^2 + list.length \end{aligned}$$

Suppose the size of a Sudoku grid is n by n, for each block (with n cells) we now have $S = n^2 + n$. For n blocks, we have $n(n^2 + n)$. Using the big O notation, the time complexity of this algorithm for n blocks is: $\mathrm{O}\big(n^3\big)$. ■

Proof. For the last formula:

$$\bigwedge_{i=1}^{|number_{row}|} \bigwedge_{j=1}^{|number_{column}|} number_{i,j} \neq 0 \rightarrow C_{i,j} = number_{i,j} \tag{5}$$

where *number* is a 2D-array, $number_{i,j}$ denotes a number that is given at the *ith* row and *jth* column, and a blank cell is represented by 0. Note that *number* is a 2D-array that has the same size as the Sudoku's grid.

One can implement Formula 5 into the the following algorithm:

Algorithm 5 Expand Formula 5

```
Input: Pre-defined values in Sudoku (stored in 2D-array $number$)
Output: $True \wedge (C_{1,1} = number_{1,1}) \wedge (C_{1,2} = number_{1,2}) \wedge ... \wedge (C_{|number_{row}|,|number_{column}|} = number_{|number_{row}|,|number_{column}|})$
Require: $Expr = True \wedge |number_{row}| \geq 1 \wedge |number_{cloumn}| \geq 1$
for $i = 1$ to $|number_{row}|$ do
    for $j = 1$ to $|number_{column}|$ do
        if $number_{i,j} \neq 0$ then
            $Expr = (Expr) \wedge (C_{i,j} = number_{i,j})$
        end if
    end for
end for
```

By analysing this algorithm, we note that variable i gets evaluated for $|number_{row}|+1$ times and j gets evaluated for $|number_{column}|+1$ times. The comparison statement (if) evaluates $number_{i,j} \neq 0$ for $|number_{column}|$ times, every time the inner loop executes. Since we assume the worse case, the assignment statement also gets evaluated for $|number_{column}|$ times. Thus, by adding them together we have:

$$
\begin{aligned}
S &= |number_{row}| + 1 + |number_{row}|(|number_{column}| + 1 + 2|number_{column}|) \\
&= |number_{row}| + 1 + |number_{row}||number_{column}| + |number_{row}| \\
&+ 2|number_{column}||number_{row}| \\
&= 3|number_{column}||number_{row}| + 2|number_{row}| + 1
\end{aligned}
$$

Suppose we use a constant n for denoting Sudoku's grid is n by n. Since the size of *number* is the same as the size of a Sudoku's grid, we now have $S = 3n^2 + 2n + 1$. By using the big O notation, the time complexity for this algorithm is: $O\big(n^2\big)$. ■

Bibliography

Marcus Alanen and Ivan Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, Turku Center for Computer Science, Finland, 2003. 18, 22

Diego Alonso, Cristina Vicente-Chicote, Pedro Sánchez, Bárbara Álvarez, and Fernando Losilla. Automatic Ada code generation using a model-driven engineering approach. In *Reliable Software Technologies –Ada Europe 2007*, volume 4498, pages 168–179. Springer, 2007. 2

Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, Nashville, TN, 2007. Springer. 18, 25, 34, 37, 137

Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010. 18, 25

Anneliese Andrews, Robert France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003. 22, 31, 91, 97

Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse.* Cambridge University Press, 2007. 20, 28

Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International*

Jounrnal on Software Tools for Technology Transfer, 11(1):69–83, Jan 2009. 20, 25, 37, 52, 137

Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003. 2

Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a large industrial context—motorola case study. In *The 8th International Conference on Model Driven Engineering Languages and Systems*, pages 476–491, Montego Bay, Jamaica, 2005. Springer. 2

Mira Balaban and Azzam Maraee. Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. *ACM Transcation on Software Engineering and Methodology*, 22(3):24:1–24:42, 2013. 6, 18, 30

Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo. 2013 sat competition. In *Proceedings of SAT Competition 2013*, 2013. 50

Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, and Gabriele Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In *7th International Conference on Fundamental Approaches to Software Engineering*, pages 214–228, Barcelona, Spain, 2004. Springer. 24

Clark Barrett and Cesare Tinelli. CVC3. In *Tthe 19th International Conference on Computer Aided Verification*, pages 298–302, Berlin, Germany, 2007. 52

Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krstić, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. The SMT-LIB Standard: Version 2.0. In *8th International Workshop on Satisfiability Modulo Theories*, Edinburgh, UK, 2010. Elsevier Science. 20, 27, 53

B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon. Genes and bacteria for automatic test cases optimization in the .NET environment. In *13th*

International Symposium on Software Reliability Engineering, pages 195–206, Annapolis, MD, 2002a. IEEE Computer Society. 23

Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components. In *17th International Conference on Automated Software Engineering*, pages 253–256, Edinburgh, UK, 2002b. IEEE Computer Society. 23

Boris Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995. ISBN 0-471-12094-4. 19

Mordechai Ben-Ari. First-order logic: Undecidability and model theory *. In *Mathematical Logic for Computer Science*, pages 223–230. Springer, 2012. ISBN 978-1-4471-4128-0. 6

Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. 34, 51

Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Automatic test data generation for XML schema-based partition testing. In *2nd International Workshop on Automation of Software Test*, Minneapolis, MN, 2007a. IEEE Computer Society. 18, 29

Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Systematic generation of XML instances to test complex software applications. In *3rd International Conference on Rapid Integration of Software Engineering Techniques*, pages 114–129, Geneva, Switzerland, 2007b. Springer. 18, 29

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *The 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer. 51

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003. 51

Georg S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2003. 6

Behzad Bordbar and Kyriakos Anastasakis. UML2Alloy: A tool for lightweight modelling of discrete event systems. In *International Conference on Applied Computing*, pages 209–216, Algarve, Portugal, 2005. IADIS. 18, 25, 34, 37, 137

Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38, 2006. 21, 38

A S Boujarwah and K Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997. 17, 21

Lionel Briand, Y. Labiche, and Q. Lin. Improving the coverage criteria of UML state machines using data flow analysis. *Software Testing, Verification and Reliability*, 20(3):177–207, 2010. 12

Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *17th International Symposium on Software Reliability Engineering*, pages 85–94, Raleigh, NC, 2006. IEEE Computer Society. 18, 22, 23, 31, 32, 34, 35, 36

Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 SMT solver. In *20th International Conference on Computer Aided Verification*, pages 299–303, Princeton, NJ, 2008. Springer. 27

Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153, Paphos, Cyprus, 2010. Springer. 27

Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003. 77

Fabian Büttner and Jordi Cabot. Lightweight string reasoning for OCL. In *The 8th European conference on Modelling Foundations and Applications*, pages 244–258, Lyngby, Denmark, 2012. Springer. 18, 142

Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *22nd International Conference on Automated Software Engineering*, pages 547–548, Atlanta, GA, 2007. IEEE Computer Society. 18, 29, 34, 138

Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL class diagrams using constraint programming. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, Berlin, Germany, 2008. IEEE Computer Society. 6, 18, 28, 29, 38, 80, 85

Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying UML/OCL operation contracts. In *7th International Conference on Integrated Formal Methods*, pages 40–55, Düsseldorf, Germany, 2009. Springer. 18, 28

Marco Cadoli, Diego Calvanese, and Toni Mancini. Finite satisfiability of UML class diagrams by constraint programming. In *2004 International Workshop on Description Logics*, British Columbia, Canada, 2004. 18, 28, 138

Marco Cadoli, Diego Calvanese, Giuseppe Giacomo, and Toni Mancini. Finite model reasoning on UML class diagrams via constraint programming. In *2007 Artificial Intelligence and Human-Oriented Computing*, volume 4733, pages 36–47. Springer, 2007. 18

Diego Calvanese. Finite model reasoning in description logics. In *The 5th International Conference on the Principles of Knowledge Representation and Reasoning*, pages 292–303, Cambridge, Massachusetts, USA, 1996. Morgan Kaufmann. 28

Diego Calvanese and Maurizio Lenzerini. On the interaction between ISA and cardinality constraints. In *The 10th IEEE International Conference on Data Engineering*, pages 204–213. IEEE Computer Society Press, 1994. 28

Felix Chang. The Alloy Analyzer 4.0. http://alloy.mit.edu/, 2007. URL `http://alloy.mit.edu/`. 20, 25, 37

S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994a. 129

S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions Software Engineering*, 20(6):476–493, 1994b. 122

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathSAT5 SMT solver. In *The 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Rome, Italy, 2013. 52

Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In *CADE-19 Workshop on Model Computation - Principles, Algorithms, Applications*, Miami, FL, 2003. 25

Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communication of the European Association of Software Science and Technology*, 24, 2009. 18

Stephen A. Cook. The complexity of theorem-proving procedures. In *The 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM. 51

L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *The 33rd International Conference on Software Engineering*, pages 331–340, 2011. 20, 25

Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *28th International Conference on Automated Software Engineering*, pages 137–148, Palo Alto, California, USA, 2009. IEEE Computer Society. 37, 52, 137

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511. 144, 145

James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *The 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, Cambridge, Massachusetts, USA, 1996a. Morgan Kaufmann. 137

James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *The 5h International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, San Francisco, USA, 1996b. Morgan Kaufmann. 137

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960. 51

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. 51

Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Budapest, Hungary, 2008. Springer. 7, 27, 52

Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, September 2011. 137

Trung. T. Dinh-Trong, Sudipto Ghosh, and B. France Robert. A systematic approach to generate inputs to test UML design models. In *17th International Symposium on Software Reliability Engineering*, pages 95–104, Raleigh, NC, 2006. IEEE Computer Society. 12

Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Adaptive star grammars. In *3rd International Conference on Graph Transformations*, pages 77–91, Natal, Rio Grande do Norte, Brazil, 2006. Springer. 24

Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13:577–625, July 2004. 30

Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *The 18th International Conference on Computer Aided Verification*, pages 81–94, Seattle, WA, USA, 2006. Springer. 52

N. Een and N. Sörensson. An Extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, Santa Margherita Ligure, Italy, 2005. Springer. 34, 51

H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, 2006. ISBN 3540311874. 38, 41, 43, 44, 45, 46, 47, 48, 49

Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer, 1985. ISBN 0387137181. 42, 45, 47, 48

Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. Generating instance models from meta models. *Software and Systems Modeling*, 8(4):479–500, 2009. 18, 24, 80, 84, 138

Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *17th International Conference on Formal Methods*, pages 133–148, Limerick, Ireland, 2011. Springer. 18, 25, 28

Maged Elaasar and Adam Neal. Integrating modeling tools in the development lifecycle with OSLC: A case study. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2013. 2

John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. *Graph Drawing*, pages 483–484, 2001. 79

Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: testing model transformations. In *First International Workshop on Model, Design and Validation*, pages 29–40. IEEE Computer Society, 2004. 18, 31, 36

Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8: 185–203, 2009. 18, 26, 32, 34, 35, 36

Ana Garis, Alcino Cunha, and Daniel Riesco. Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In *9th International Conference on Software Engineering and Formal Methods*, pages 221–236, Montevideo, Uruguay, 2011. Springer. 18, 25

Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, 4:386–398, 2005. 18, 29, 34, 35

Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007. 29, 35

Ulrike Golas. *Analysis and correctness of algebraic graph and model transformations.* PhD thesis, Berlin Institute of Technology, 2011. 43

Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, June 2007. 51

Carlos Alberto González Pérez, Fabian Buettner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches*, Zurich, Suisse, 2012. 18, 29, 34, 85, 138

Jörg Harm and Ralf Lämmel. Testing attribute grammars. In *3rd Workshop on Attribute Grammars and their Applications*, pages 79–99, Ponte de Lima, Portugal, 2000. 21

Reiko Heckel and Leonardo Mariani. Automatic conformance testing of web services. In *8th International Conference on Fundamental Approaches to Software Engineering*, pages 34–48, Edinburgh, UK, 2005. Springer. 18

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and Java. In *2nd International Conference on Software Language Engineering*, pages 374–383, Denver, CO, 2010a. Springer. 12, 113

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and Java. In *2nd International Conference on Software Language Engineering*, pages 374–383, Denver, CO, 2010b. Springer. 2

Berthold Hoffmann and Mark Minas. Defining models - meta models versus graph grammars. *Electronic Communications of the EASST*, 29:1–14, 2010. 19

Berthold Hoffmann and Mark Minas. Generating instance graphs from class diagrams with adaptive star grammars. In *3rd International Workshop on Graph Computation Models*, 2011. 18, 24

ILOG. *ILOG Solver system version 5.1 user's manual.* IBM, 2001. 28

ILOG. *ILOG OPL Studio system version 3.6.1 user's manual.* IBM, 2002. 28

D. Jackson and C.A. Damon. Elements of style: analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996. 65, 84

Daniel Jackson. An intermediate design language and its analysis. In *6th International Symposium on Foundations of Software Engineering*, pages 121–130, Lake Buena Vista, FL, 1998. ACM. 34, 84

Daniel Jackson. Automating first-order relational logic. In *The 8th International Symposium on Foundations of Software Engineering*, pages 130–139, San Diego, CA, 2000. ACM. 20, 34

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, 2002. 18, 25, 34, 84

Daniel Jackson. *Software Abstractions: logic, language and analysis.* MIT Press, 2006. 25

Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: a new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, March 1998. 25

Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the Alloy constraint analyzer. In *International Conference on Software Engineering*, pages 730–733, Limerick, Ireland, 2000. ACM. 34

Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. *ACM SIGSOFT Software Engineering Notes*, 26(5):62–73, September 2001. 25

Ethan Jackson, Tihamér Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *14 International Conference on Model Driven Engineering Languages and Systems*, pages 653–667, Wellington, New Zealand, 2011. Springer. 18, 27, 34, 37

Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using UML for software process modeling. In *7th European Software Engineering Conference*, pages 91–108. Springer, 1999. 2

Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Conflict anticipation in the search for graph automorphisms. In *The 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 243–257, Mérida, Venezuela, 2012. Springer. 137

Barbara Kitchenham. Procedures for performing systematic reviews. Technical report, Keele University, 2004. 9

Mirco Kuhlmann and Martin Gogolla. Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012. 18, 25, 26

Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *49th International Conference on Objects, Models, Components, Patterns*, pages 290–306, Zurich, Switzerland, 2011. Springer. 18, 34, 35, 137

Jochen M. Küster and Mohamed Abd-El-Razik. Validation of model transformations: first experiences using a white box approach. In *9th International Conference on Models in Software Engineering*, pages 193–204, Genoa, Italy, 2006. Springer. 17

Maher Lamari. Towards an automated test generation for the verification of model transformations. In *22nd Symposium on Applied Computing*, pages 998–1005, Seoul, Korea, 2007. ACM. 18, 36

Ralf Lämmel. Grammar testing. In *4th International Conference on Fundamental Approaches to Software Engineering*, pages 201–216, Genova, Italy, 2001. Springer. 22

Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *18th International Conference on Testing of Communicating Systems*, pages 19–38, New York, USA, 2006. Springer. 22

Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall, 2003. 2

Wei Li and Sallie M. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993. 122

Tomaž Lukman, Marjan Mernik, Zekai Demirezen, Barrett Bryant, and Jeff Gray. Automatic generation of model traversals from metamodel definitions. In *48th*

Annual Southeast Regional Conference, pages 78:1–78:6, Oxford, Mississippi, USA, 2010. ACM. 18, 30

Den Haag M. Feenstra. Sudoku. http://www.sudoku.ws/extreme.htm, August 2013. 55

Nuno Macedo and Alcino Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In *The 16th International Conference on Fundamental Approaches to Software Engineering*, pages 297–311. Springer, Rome, Italy, 2013. 19, 34, 36, 37, 86, 138

Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: an efficient SAT solver. In *The 7th International Conference on Theory and Applications of Satisfiability Testing*, pages 360–375, Trento, Italy, 2005. Springer. 51

Alireza Mahdian, Anneliese Amschler Andrews, and Orest Pilskalns. Regression testing with UML software designs: a survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(4):253–286, 2009. 12

Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009. 137

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. 120, 127

Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30: 45–87, 1981. 137

Jacqueline A. McQuillan and James F. Power. A metamodel for the measurement of object-oriented systems: An analysis using Alloy. In *1st International Conference on Software Testing Verification and Validation*, pages 288–297, Lillehammer, Norway, 2008. IEEE Computer Society. 18, 25, 26, 34, 35, 37

Aleksandar Milicevic and Hillel Kugler. Model checking using SMT and theory of lists. In *The 3rd International Conference on NASA Formal Methods*, pages 282–297, Pasadena, CA, USA, 2011. Springer. 52, 137

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference*, pages 530–535, Las Vegas, Nevada, United States, 2001. ACM. 34, 51

Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform random generation of huge metamodel instances. In *5th Europrean Conference on Model Driven Architecture - Foundations and Applications*, pages 130–145, Enschede, The Netherlands, 2009. Springer. 18, 30, 34, 36

Ashalatha Nayak and Debasis Samanta. Model-based test cases synthesis using UML interaction diagrams. *Software Engineering Notes*, 34(2):1–10, 2009. 12

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 36–50. Springer, 2005. 52

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis-putnam-logemann-loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006. 52

Object Management Group. Meta Object Facility Core Specification v2.4.1, August 2011a. 2, 4, 12, 21

Object Management Group. Meta Object Facility 2.0 Query/View/Transformation/Specification Version 1.1, January 2011b. 19

Object Management Group. Unified Modeling Language, Infrastructure Version 2.4.1, August 2011c. 4, 12

Object Management Group. Unified Modeling Language, Superstructure Version 2.4.1, August 2011d. 4, 30

Object Management Group. Object Constraint Language Version 2.3.1, Jan 2012a. 35

Object Management Group. Object Constraint Language Version 2.3.1, January 2012b. 4

T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988. 22, 29

Stephan Philippi. Automatic code generation from high-level Petri-Nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444–1455, 2006. 2

Orest Pilskalnsa, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France. Testing UML designs. *Information and Software Technology*, 49(8):892–912, 2007. 12

B. Poonen. Undecidable problems: a sampler. *ArXiv e-prints*, 2012. 38

Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In *2004 Principles and Practice of Constraint Programming*, volume 3258, pages 5–8. Springer, 2004. 38, 137

P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972. 10, 21

Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, 73:1–22, 2012. 18

Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1. Foundations.* World Scientific Publishing, 1997. ISBN 98-102288-48. 19

Philip Samuel and A.T. Joseph. Test sequence generation from UML sequence diagrams. In *9th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 879–887. IEEE Computer Society, 2008. 12

Sagar Sen and Benoit Baudry. Mutation-based model synthesis in model driven engineering. In *2nd Workshop on Mutation Analysis*. IEEE Computer Society, 2006. 18, 36

Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *1st International Conference on Software Testing, Verification, and Validation*, pages 328–337, Lillehammer, Norway, 2008. IEEE Computer Society. 18, 25, 26, 37

Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *2nd International Conference on Theory and Practice of Model Transformations*, pages 148–164. Springer, 2009. 18, 25, 26, 34, 35, 37

Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to alloy and back again. In *6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 4:1–4:10. ACM, 2009. 25

Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, June 2007. 115, 137

Michael Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997. ISBN 978-0-534-94728-6. 144, 145

M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation Test in Europe Conference Exhibition*, pages 1341–1344, Dresden, Germany, 2010. 18, 27, 136

Mathias Soeken, Robert Wille, and Rolf Drechsler. Towards automatic determination of problem bounds for object instantiation in static model verification. In *8th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 2:1–2:4, Wellington, New Zealand, 2011a. ACM. 18, 27

Mathias Soeken, Robert Wille, and Rolf Drechsler. Encoding OCL data types for SAT-based verification of UML/OCL models. In *5th International Conference on Tests and Proofs*, pages 152–170, Zurich, Switzerland, 2011b. Springer. 18, 27, 37, 84, 136, 137, 138, 140, 142

David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2008. 105

Mana Taghdiri Tianhai Liu, Michael Nagel. Bounded program verification using an SMT solver: A case study. In *IEEE 5th International Conference on Software Testing, Verification and Validation*, pages 101–110, Montreal, QC, Canada, 2012. IEEE Computer Society. 25, 37, 52, 137

Emina Torlak and Daniel Jackson. The design of a relational engine. Technical Report MIT-CSAIL-TR-2006-068, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2006. 18, 25

Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Braga, Portugal, 2007. Springer. 18, 25, 34, 137

Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *The 15th International Symposium on Formal Methods*, pages 326–341, Turku, Finland, 2008. Springer. 25

Eminate Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2009. 25, 26, 34, 37, 51, 137, 138

G S Tseitin. On the complexity of derivation in propositional calculus. *Studies in Mathematics and Mathematical Logic*, 2:115–125, 1968. 20

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing, 1999. 23

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. ISBN 0321179366. 80

R. Wille, M. Soeken, and R. Drechsler. Debugging of inconsistent UML/OCL models. In *2012 Design, Automation Test in Europe Conference Exhibition*, pages 1078–1083, 2012. 18

Jessica Winkelmann, Gabriele Taentzer, Karsten Ehrig, and Jochen M. Küster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *Electronic Notes in Theoretical Computer Science*, 211:159–170, 2008. 18, 24

World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008. 21

Hao Wu, Rosemary Monahan, and James F. Power. Test case generation for programming language metamodels. In *Doctoral Symposium of the 3rd International Conference on Software Language Engineering*, Eindhoven, Netherlands, 2010. 6

Hao Wu, Rosemary Monahan, and James F. Power. Metamodel instance generation: A systematic literature review. *Computing Research Repository*, abs/1211.6322, 2012. 7

Hao Wu, Rosemary Monahan, and James F. Power. Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *7th International Symposium on Theoretical Aspects of Software Engineering*, Birmingham, UK, 2013. 6, 7, 68, 138

Kenro Yatake and Toshiaki Aoki. SMT-based enumeration of object graphs from UML class diagrams. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, July 2012. 18, 80

T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E86-A(5):1052–1060, 2003. 55