Formal Verification of Relational Model Transformations using an Intermediate Verification Language

Zheng Cheng

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

Department of Computer Science

Maynooth University

This dissertation is submitted for the degree of

Doctor of Philosophy

Sept 25, 2015

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text. This dissertation contains less than 65,000 words including appendices, bibliography, footnotes, tables and equations and has less than 150 figures.

Zheng Cheng
Sept 25, 2015

Acknowledgements

First and foremost, I want to thank my parents for all their help and support throughout my years of study. Without them I would never have been able to undertake, or complete a Ph.D. This thesis is dedicated to them.

I am extremely grateful and fortunate to have worked with my supervisors, Rosemary Monahan and James F. Power, whose expertise, understanding, and patience, added considerably to my graduate experience. I appreciate their vast knowledge and skill in many areas (e.g. program verification and compilation), and their constant assistance in writing reports. I am also grateful to Dominique Méry, Shmuel Tyszberowicz, the members of the Principles of Programming research group and our department, both past and present, for many helpful discussions of computer science and philosophy.

Finally, I want to thank my families for always have faith in me, and my friends Fangzhou, Han, Hao, Kaijie, Lingfei, Long, Mingming, Xu, Yifan and Ziming for getting me through those good and bad days.

Acronym

Abbr.	Phrase	Debut (Pg.)
ASM	ATL Stack Machine	6
ATL	Atlas Transformation Language	2
CPN	Coloured Petri-nets	13
CSP	Constraint Satisfaction Problem	12
EMF	Eclipse Modelling Framework	2
EMFTVM	EMF Transformation Virtual Machine	7
FOL	First Order Logic	4
GT	Graph Transformation	2
IVL	Intermediate Verification Language	4
MDE	Model Driven Engineering	1
MTr	relational Model Transformation	2
OCL	Object Constraint Language	5
OMG	Object Management Group	1
OO	Object Oriented	5
QVTr	Query/View/Transformation relational	15
SAT	Bounded Boolean Satisfiability	12
SMT	Satisfiability Modulo Theories	12
UML	Unified Modelling Language	1
VC	Verification Condition	3
TGG	Triple Graph Grammar	16

Abstract

Model-driven engineering has been recognised as an effective way to manage the complexity of software development. Model transformation is widely acknowledged as one of its central ingredients. Among different paradigms of model transformations, we are specifically interested in relational model transformations.

Proving the correctness of relational model transformations is our major concern. Typically "correctness" is specified by MTr developers using contracts. Contracts are the annotations on the MTr which express constraints under which the MTr are considered to be correct. Our main objective is to develop an approach to designing a deductive verifier in a modular and sound way for a given target relational model transformation language, which enables formal verification of the correctness of MTr.

To this end, we have developed the VeriMTLr framework. Its role is to assist in designing verifiers that allow verification (via automatic theorem proving) of the correctness of relational model transformations. VeriMTLr draws on the Boogie intermediate verification language to systematically design modular and reusable verifiers for a target relational model transformation language. Our framework encapsulates an EMF metamodels library and an OCL library within Boogie. The result is reduced cost and time required for a verifier's construction. Furthermore, VeriMTLr includes an ASM and EMFTVM bytecode library, enabling an automated translation validation approach to ensuring the soundness of the verification of the designed verifier. We demonstrate our VeriMTLr framework with the design of verifiers for the Atlas Transformation Language and the SimpleGT graph transformation language.

List of Publications

Proposal	Cheng, Z. (2012). A Proposal for a Generic Translation Framework for Boogie Language. *26th European Conference on Object-Oriented Programming (Doctoral Symposium)*, Beijing, China.
Overview	Cheng, Z. (2015). Formal Verification of Relational Model Transformations using an Intermediate Verification Language. *3rd International Conference on Model-Driven Engineering and Software Development (Doctoral Consortium)*, Angers, France.
Chapter 3,4,5	Cheng, Z., Monahan, R., and Power, J. F. (2015). A sound execution semantics for ATL via translation validation. *In 8th International Conference on Model Transformation*, pages 133–148, L'Aquila, Italy. Springer.
Chapter 6	Cheng, Z., Monahan, R., and Power, J. F. (2015). Verifying SimpleGT Transformations Using an Intermediate Verification Language. *In 4th International Workshop on the Verification Of Model Transformation*, To Appear, L'Aquila, Italy. CEUR.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the late 20th century, the Object Management Group (OMG) envisioned a new paradigm, called Model Driven Engineering (MDE), to address the full life cycle of software development [93]. MDE is seen as an evolution from the earlier object management architecture or component-oriented technology. It focuses on accurately modelling the problem rather than programming it, which allows the problem to be well-comprehended before generating an implementation. In addition, MDE unifies some of the best practices in software architecture, including modelling, meta-data management and model transformation technologies. Thus, it allows a user to model once and to target multiple technology implementations by using precise model transformations. MDE is often discussed in conjunction with several closely related concepts such as models, metamodels and transformations (e.g. relational, operational and graph model transformations).

Models and Metamodels. A model is a simplification of a system built with an intended goal in mind, which should be able to answer questions in place of the actual system [14]. The simplification (or abstraction) is the essence of the modelling: depending on a specific task, many unnecessary aspects from the original system are abstracted away, and only a few are preserved in the resulting model. Thus, understanding the model of the system should be easier than understanding the original system.

A metamodel is the specification for a model [14]. The metamodel introduces the structure for expressing the models (as in a context-free grammar for programming languages). One could say a model **conforms** to its metamodel when the model meets its specification. A well-known metamodelling language is the Unified Modelling Language (UML) which was proposed by the OMG group [102]. UML emphasises the idea of using different views to model different aspects of complex systems, as no single view can capture all aspects of the system completely. Therefore, it includes several languages which describe different aspects of the system, e.g. the class diagram for structural modelling or the activity diagram for behavioural modelling.

These modelling and metamodelling concepts are brought into practice through the Eclipse Modelling Framework (EMF). EMF draws on the facilities provided by Eclipse, to link modelling with programming [109]. Not only does EMF provide tools and run-time support to translate the model into a Java implementation, but also brings with it other facilities that enable viewing and editing of the models.

Model Transformation. Model transformation is widely acknowledged as one of the central ingredients of MDE. A model transformation is the automatic generation of a target model from a source model, according to a transformation specification [69]. Three main paradigms for developing model transformations are the operational, relational and graph-based approaches [38]:

- Operational model transformations, such as Kermeta [65] and operational QVT [94], are imperative in style, and focus on imperatively specifying **how** a model transformation should progress.
- Relational Model Transformations (MTr), such as Atlas Transformation Language (ATL) [66] and ETL [70], have a "mapping" style, and aim to produce a declarative transformation specification that documents **what** the model transformation intends to do. Typically, a declarative specification is compiled into a low level transformation implementation and is executed by the underlying virtual machine.
- Graph transformations (GT), such as SimpleGT [118] and Henshin [4], use a rewriting style, which applies transformation rules recursively until no more matches can be found.

1.1 Research Objective

Our overall objective in this research is:

To develop an approach to design a deductive verifier in a modular and sound way for a given target MTr language, which enables formal verification of the correctness of MTr.

Some explanation of our objective is in order. First, the correctness of MTr is our major concern in this research. Typically "correctness" is specified by MTr developers using contracts. Contracts are the annotations on the MTr which express assumptions under which condition the MTr are considered to be correct.

With the increasing complexity of model transformation (e.g. automotive industry [107], medical data processing [117], aviation [12]), it is urgent to develop techniques and tools that prevent incorrect model transformations from generating faulty models. The effects of such faulty models could be unpredictably propagated into subsequent MDE steps, e.g. code generation, to produce further errors.

We are specifically interested in MTr. Because of its declarative nature, a MTr is generally easier to write and understand than an operational transformation. Both graph transformations and relational transformations can be declarative. However, they are essentially different in their rule matching and execution semantics [38]. In addition, deciding the confluence and termination of graph transformations has been proven undecidable [95, 96]. This extra layer of complexity promotes the applicability of MTr over the graph transformations.

Second, the purpose of verifying MTr correctness is to ensure that the MTr is built in a way that matches the assumptions of developers [17]. In contrast to validation, verification does not involve reasoning about the validity of assumptions of developers.

Third, formal verification ensures that the verification tasks are guided by formal methods of mathematics, which distinguishes itself from informal approaches such as testing. In addition, we strive for sound formal verification in order to prevent false negatives. Ab.Rahim and Whittle, in their survey, find that one of the main challenges in the research on model transformation verification is reasoning about the soundness of the proposed verification approach, which is an under-researched area in MDE [1].

Fourth, we aim for deductive formal verification. This emphasises the use of logic to prove the correctness of MTr. Thus, it distinguishes itself from other formal verification approaches (e.g. model-based, or type-based formal verification). Typically, a verifier needs to be designed for this task. The verifier will be guided by the chosen logic (e.g. Hoare Logic [58]) to translate the MTr which is to be verified, as well as its contracts, into a set of logical formulas known as Verification Conditions (VCs). These formulas are then interpreted and processed by a theorem prover. A failed proof indicates a mismatch between the MTr and its contracts, whereas a successful proof indicates the correctness of the MTr in terms of its contracts. Consequently, the users of the verifier are able to verify the correctness of MTr without actually running it, thereby reducing the time for quality assurance and enhancing productivity. Moreover, there is no need to test the MTr against a particular source model after deductive verification, since its correctness is proved in general, thereby holding for all the possible source models. However, we will gradually show in the following chapters that deductive verification is a challenging task, which requires experience and creativity to build the proof.

Finally, building a verifier is a non-trivial task [8]. That is why we are interested in modular design, to allow the effort or knowledge to be reused.

1.2 Research Problems

Based on the overall objective of our research, we identify two research problems.

Research Problem 1. *Our quest is to investigate whether using an Intermediate Verification Language (IVL) is the most suitable approach to systematically designing modular and reusable verifiers for a target MTr language.*

Previous program verifier designs have already established the usefulness of an IVL in decomposing the complex task of generating VCs for general high-level programming languages into two steps [7, 50, 82]: a transformation from the program and its proof obligations into the program written in an IVL, and then a transformation from an IVL program into logical formulas. Thus, the IVL bridges between the front-end high-level programming language and the back-end theorem prover. The benefit is that it allows us to focus on generating proof obligations that prescribe what correctness means for the front-end language in a structural way, and to delegate the task of interacting with theorem provers to the IVL.

The two most widely used IVLs are Boogie [7], and Why3 [52]. Both of them are based on First Order Logic (FOL) with polymorphic types, and have mature implementations to parse, type-check, and analyse programs. We concentrate on Boogie in this research, but we believe all results can be reproduced in Why3 (a coarse roadmap is demonstrated in Appendix E) or other IVLs with comparable functionality.

At the time of this thesis being written, verifier designs for high-level programming languages that target Boogie exist for C# [8], C [39], Dafny [82], Java [80], and Eiffel [114]; verifier designs that target Why3 exist for Java [90], C [36], Ada [57], and B [45].

However, we find that it is difficult to design a new verifier based on the effort and knowledge obtained from the existing verifier designs. We think this is mainly because the paradigms and support features of general high-level programming languages differ from each other.

Thus, the intuition behind our first research problem is that by limiting our focus to a set of more task-specific programming languages (i.e. MTr languages), we should be able to verify the program correctness in an integral way, taking into account the modularity and reusability of the verifier design.

Research Problem 2. *We investigate whether the translation validation approach from compiler verification can be automated to ensure the soundness of the verification, i.e. to be able to check consistency between the execution semantics of each transformation specification and the runtime behaviour of its corresponding transformation implementation.*

Translation validation is a technique used in compiler verification to ensure each individual compilation is followed by a validation phase which verifies that the compiled program correctly implements the source program [97].

For MTr, recall that to be executable, the transformation is usually compiled into low-level bytecode. We believe it is intuitive and feasible to adopt the translation validation approach to ensure the soundness of the verifier.

However, as our second research problem suggests, we also anticipate changes to be made while adopting the translation validation approach. This is mainly because of the domain-specific properties that are only present in compiling an MTr, e.g. the bytecode instructions that are specifically designed for model handling. In addition, another layer behind the second research problem is that we aim to automate the translation validation approach in the domain of MTr, since translation validation is usually interactively proved for general programming languages (e.g. the Compcert project, a verified compiler for a large subset of C [86]).

Our systematic literature review in Chapter 2 suggests that both identified research problems have not been addressed before. Thus, it is our desire to provide answers to them through this research.

1.3 Overview of Contributions

The contributions of this thesis are split into four parts. First, we detail the semantics of the EMF metamodels and Object Constraint Language (OCL) in Chapter 3 [32]. Both of these are encoded in the Boogie IVL as libraries and intended for reuse across different verifier designs for MTr. In particular:

1. We adapt the formalisation of Object Oriented (OO) programs (specifically its memory model) to formalise the semantics of EMF metamodels for expressing the metamodels involved in a MTr. This is based on the observation that the concepts of metamodelling share many similarities with OO programming language constructs,

but with subtle differences. The implications of our adaptation are twofold. First, because of the shared similarities with OO constructs, the metamodel formalisation will be easy to comprehend and reuse. Second, the chosen memory model has been used by several program verifiers (e.g. Spec# [8], KIV [110]). Therefore, it will enhance interoperability between verifiers.

2. We extend the existing Boogie libraries from the Dafny verification language [82], to develop the semantics of OCL for expressing transformation contracts. This further demonstrates reusability due to adopting an IVL, i.e. we are able to build on top of an existing Boogie library, and the libraries we developed are also made available to others for reuse.

The second part of this thesis focuses on the design of the VeriATL system using the Boogie IVL to enable the verification of partial correctness for the ATL language (Chapter 4) [32]. This shows the feasibility of using an IVL to systematically design modular and reusable verifiers for a target MTr language. The core component of VeriATL is the execution semantics of ATL, which is formalised based on the two modular Boogie libraries for the semantics of EMF and OCL. The details were far from obvious to us, and articulating them is one of the main challenges of developing VeriATL. In particular:

1. We experiment with the use of separate memory models to simplify the encoding for the execution semantics of ATL.

2. Furthermore, we illustrate the use of frame conditions at field granularity to precisely capture how the states of MTr are evolved.

The third part of this thesis is motivated by a potential unsoundness in our encoded execution semantics of ATL. Therefore, we present an automated translation validation approach to ensure the encoding soundness (Chapter 5) [32]. This is achieved by certifying that the encoded execution semantics for ATL soundly represents the runtime behaviour of its corresponding ATL Stack Machine (ASM) bytecode implementation. Consequently, we are confident that the verification of partial correctness of ATL that is based on our sound encoding will be sound too. Our translation validation approach is modular. We compositionally verify the soundness of our Boogie encoding for the execution semantics of each ATL matched rule. This is based on the semantics of the ASM bytecode language:

1. We give a translational semantics of the ASM language via a list of translation rules. Each rule encodes the operational semantics of an ASM bytecode in Boogie. Consequently, we are able to use such translation semantics to precisely explain the runtime behaviour of ASM implementations. The challenge is to handle the versatility of the ASM language against different model management systems. Our strategy is to focus on the EMF model management system (which represents a de facto standard for modelling nowadays), and to investigate the source code that is specific to it.

2. We observe the characteristics of loops inside the ASM implementation. Thus, we are able to automatically generate variant expressions inside loops to ensure the termination of ATL transformation at runtime, thereby ensuring its total correctness.

Finally, the fourth part of this thesis is dedicated to investigating the possibility of reusing our previous efforts to design a modular and reusable verifier for a GT language, namely SimpleGT (Chapter 6) [33]. In particular:

1. We demonstrate the differences between the execution semantics of relational and graph transformations, and quantify how the differences would affect their verifier designs.

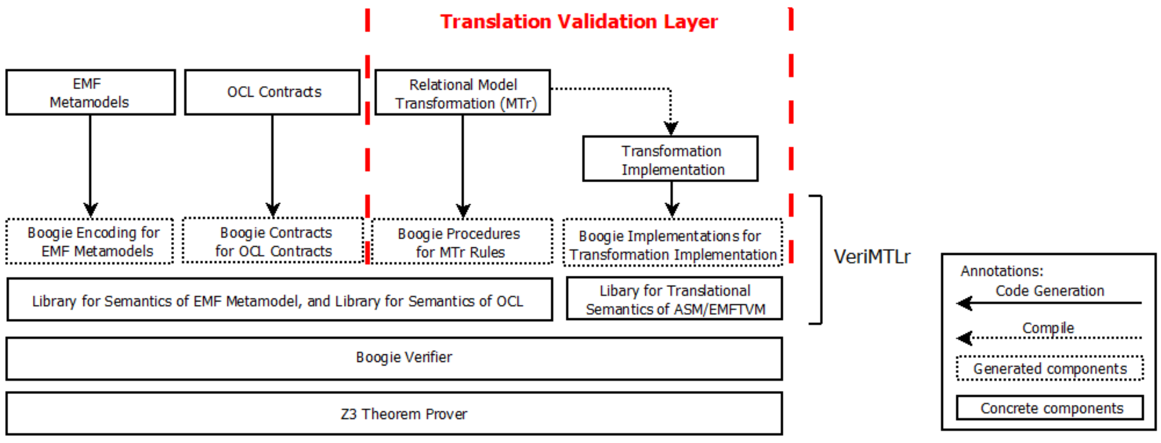
2. We also illustrate how to develop the semantics of the EMF Transformation Virtual Machine (EMFTVM) language by extending the semantics of the ASM language, enabling translation validation for a wider range of model transformation languages, especially for those with explicit memory deallocation.

We have captured the semantics of the EMF metamodel, OCL, ASM and EMFTVM as libraries in the Boogie IVL. These libraries are under the hood of our VeriMTLr framework (Fig. 1.1), which allows them to be reused to provide rapid verifier construction for MTr languages.

Nomenclature. We refer to the annotations that are explicitly made by transformation developers to express transformation correctness as *contracts*. This is because in the context of MDE, the term *specification* usually refers to a transformation program [69].

Throughout the thesis, we do not make any distinction between the terms *verifier*, and *verification system*. Each refers to software that performs deductive verification tasks. We use these terms interchangeably.

Boogie is the name of an IVL and a verifier. To avoid name conflict, we use *Boogie* to refer to the Boogie IVL, and use *Boogie verifier* to refer to the verifier for Boogie.



Translation Validation Layer
EMF Metamodels
OCL Contracts
Relational Model Transformation (MTr)
Transformation Implementation
Boogie Encoding for EMF Metamodels
Boogie Contracts for OCL Contracts
Boogie Procedures for MTr Rules
Boogie Implementations for Transformation Implementation
VeriMTLr
Library for Semantics of EMF Metamodel, and Library for Semantics of OCL
Libary for Translational Semantics of ASM/EMFTVM
Boogie Verifier
Z3 Theorem Prover
Annotations:
Code Generation
Compile
Generated components
Concrete components

Fig. 1.1 The architecture of our VeriMTLr development framework

To improve the readability of the Boogie code in this thesis, we use a list of symbols to denote the sequence operations. Specifically, let *S* represents a sequence. |S| denotes the length of *S*. e ∈ S denotes *e* is contained by *S*. [] denotes a sequence constructor, e.g. [] is an empty sequence, and [e] is a sequence that contains *e*. S[i] denotes the element at the index *i* of *S*.

Chapter 2

Literature Review

The goal of this chapter is to find the most reliable and practical approach to enable formal verification of MTr. First, we identify four literature review questions related to the formal verification of MTr (Section 2.1). Next, we systematically review and analyse the literature to provide answers to these questions (Section 2.2 and Section 2.3). By answering these questions, we obtain a clear perspective of the advantages and disadvantages of the state of the art (Section 2.4). This will guide us to propose our approach for the formal verification of MTr (Section 2.5).

2.1 Literature Review Questions

Although some literature reviews have been conducted to analyse the state of the art in the field of model transformations verification [1, 2, 26], these studies are broad and general, and do not specially fit the focus of our research: they neither restrict the analysis to those approaches applied over MTr, nor focus on reviewing the utilisation of different formal methods/formalisation techniques to verify model transformations. Moreover, these studies do not categorise how to express transformation contracts. Thus, we propose the following four literature review questions, and will analyse them in the next sections:

- LRQ1: What formal methods are employed in MTr verification?
- LRQ2: What formalisms are employed in MTr verification?
- LRQ3: What transformation contracts are verified? How are they expressed?
- LRQ4: To what extent are the existing approaches supported by a tool?

2.2 Exploration Method

To answer the identified literature review questions, we adopt a snow-ball approach to systematically retrieve references related to the formal verification of MTr published before 2014 [119]. Specifically, we identify three literature reviews that analyse the research on model transformation verification published before 2012 [1, 2, 26]. Then, from the references of the identified literature reviews, the following steps are applied in order to filter out the references that are not specific to the formal verification of MTr.

1. Delete duplicate references.
2. Delete references with no author.
3. Pruning by title.
4. Pruning by keyword.
5. Pruning by abstract.

The following inclusion and exclusion criteria are used in the pruning steps:

- (Inclusion) Specific to MTr.
- (Inclusion) Specific to formal methods.
- (Exclusion) Specific to testing.
- (Exclusion) Not written in English.

If a reference remains unclear for its relevancy to the formal verification of MTr after the pruning steps, it is thoroughly read. In this way, we identify a initial set of references that are specific to the formal verification of MTr. Next, the Google Scholar search engine[1] is used to find new references published after 2012 that refer to any references in the initial set. Google Scholar is chosen as it indexes most of the digital reference libraries. The same inclusion and exclusion criteria are applied to the newly found references. The whole process is iterated until no more new references that were published before 2014 are found.

Limitations. Admittedly, our literature review has the possibility of missing some relevant references due to the following limitations:

[1]Google Scholar search engine. http://scholar.google.com.

- First, our exploration method focuses on international workshops/conferences/journals whose communication is typically via English. It is possible that we miss references that are not written in English, e.g. references that are presented at national workshops/conferences/journals, or presented as technical reports in native languages.
- Second, we chose the Google Scholar search engine to enrich the references rather than the digital reference libraries. It is not guaranteed that every reference indexed by the digital reference libraries will be also indexed by Google Scholar.
- Third, the three identified literature reviews together present a relative complete study about works regarding model transformation verification published before 2012. However, it is still possible that certain references are not included in these literature reviews.

In spite of the limitations identified here, we are confident that we did not miss a large number of relevant references, and the conclusions we derive in this chapter are still valid.

2.3 Data Extraction

Our exploration yields 30 references regarding formal verification of MTr. We are now in the position to answer the posed literature review questions LRQ1-LRQ4. In addition, we summarise the answers to our literature review questions in Table 2.1 at the end of this section.

2.3.1 LRQ1: What Formal Methods are Employed in MTr Verification?

From all the studies analysed, we conclude that there are three kinds of formal methods applied in MTr verification.

- Simulation [56, 113, 120].
- Model checking and model finding [3, 9, 22–24, 47, 63, 87, 88, 105, 106, 111].
- Theorem proving [21, 25, 28, 35, 49, 72–79, 98, 99].

Simulation. The first category of MTr verification builds on simulation techniques. Simulation techniques generally require a mathematical model to be developed. This mathematical model represents the key characteristics of the MTr (e.g. source and target metamodels, the behaviour of the MTr transformation). Next, a simulation tool is used to simulate the mathematical model against a particular input (which is developed from a given source model). Depending on the chosen tool, certain kinds of correctness can be expressed as contracts, and can then be verified for the chosen input (Section 2.3.3).

Model checking and model finding. The second category concerns the model checking and model finding techniques. Similar to simulation techniques, model checking and model finding techniques also require a mathematical model to be developed (from the metamodels and the MTr transformation). However, compared to simulation techniques, no particular input is needed when the model checking/finding is running.

Generally, model finding techniques are very similar to model checking techniques. However, a subtle difference between them is the way in which the developed mathematical models are used [61]. The former starts with a mathematical model described by the user, and it discovers whether the contracts asserted by the user are valid on the mathematical model. The latter finds mathematical models which form counter-examples to the contracts made by the user.

Model finding techniques consist of three main sub-categories: bounded Boolean Satisfiability (SAT)-based, Satisfiability Modulo Theories (SMT)-based and Constraint Satisfaction Problem (CSP)-based model finding. The general idea of SAT-based model finding is to reduce the model transformation verification problem into a SAT problem[2]. The source and target metamodels, the transformation, and the transformation contracts are encoded into a SAT formula. Then, this formula is sent to a SAT solver, along with a predefined search space. The goal is to ask a SAT solver to find a counter-example (i.e. a case where transformation contracts do not hold on the given transformation) within the given search space. If any counter-example is found, the model transformation is not verified. However, if no counter-example is found, no conclusion can be drawn (e.g. a counter-example could be found in a larger search space).

SMT-based model finding is another important technique for performing MTr verification. The general idea is to use a SMT solver in the model finding process. SMT solvers are an extension to SAT solvers with built-in background theories for real numbers, integers, data structures (e.g. bit vectors and arrays) and so on. The major advantage of SMT solvers is the enhanced expressiveness (i.e. FOL with equality) to handle constraints over infinite domains.

[2]The SAT problem is to determine whether the variables of a given Boolean formula can be consistently replaced by the values *TRUE* or *FALSE* in such a way that the formula evaluates to TRUE.

CSP-based model finding is an alternative to SAT-based and SMT-based model finding, where a CSP solver (that is neither SAT-based nor SMT-based) is used as the model transformation verification engine.

Theorem proving. Theorem proving is a deductive formal verification approach. It focuses on formalising both the MTr transformation and its contracts into formulas. Verification consists of applying deduction rules (of an appropriate logic) to incrementally build the proof. Theorem proving can be performed by using either pen and paper or specific proving tools (e.g. Coq [13]). It also does not require a specific source model during the proof.

In conclusion, simulation, model checking and model finding are model-based verification techniques, whereas theorem proving is proof-based [61]. In the model-based approach, the metamodels and MTr are formalised into a mathematical model for an appropriate logic. The contracts are formalised into a formula of a compatible logic. Then, the verification consists of computing whether a mathematical model satisfies its contracts in the prover that supports reasoning in the chosen logic. This computation is usually automatic for finite mathematical models. In contrast, the proof-based approach formalises both the MTr and its contracts as a formula. The verification consists of applying deduction rules (of an appropriate logic) to incrementally build the proof for the derived formula, which usually requires guidance and expertise from the user.

2.3.2 LRQ2: What Formalisms are Employed in MTr Verification?

In all analysed references, formalisation is used when transforming metamodels, transformation and optionally transformation contracts into the formalism that is used to perform MTr verification. In this regard, the most typical formalism used in MTr verification is to use a logical representation. FOL is the most popular one [21–24, 47, 72, 75–78, 105]. Other logical representations have also been used, but are less common, e.g. relational logic (i.e. FOL plus additional relational operators) [3], and rewriting logic [113]. Contract languages like B and Coloured Petri-nets (CPN) are alternative formalisms used in MTr verification [56, 79, 120]. Finally, it is also possible to encode MTr verification by means of mathematical notations such as graph theory [63, 87, 88, 106], type theory [28, 49, 98, 99] and game theory [111].

2.3.3 LRQ3: What Transformation Contracts are Verified? How are they Expressed?

The transformation contracts verified by all the identified references fall into 6 categories:

(C1) **Syntactic correctness** of MTr ensures that every valid source model generates a valid target model. The validity is usually given by syntactic constraints, e.g. constraints on the multiplicity of associations [3, 21–25, 47, 49, 63, 72, 75–79, 87, 88, 98, 99, 105, 106].

(C2) **Semantic correctness** of MTr ensures that semantic constraints (e.g. constraints on the uniqueness of associations) defined on the source metamodel, can be preserved on the target metamodel after executing the MTr [21–23, 35, 47, 49, 72, 75–79, 98, 99, 105, 106, 111].

(C3) **Semantic preservation** of MTr ensures that the formalised execution semantics of a MTr soundly represents the runtime behaviour of its corresponding transformation implementation.

(C4) **Confluence** of MTr ensures that a MTr generates the same target models from the same source models [56, 87, 113, 120].

(C5) **Termination** of MTr ensures that a MTr terminates under valid source models [56, 87, 120].

(C6) **Transformation quality** of MTr ensures that the MTr is specified with certain qualities [56, 113, 120], such as:

- Absence of rule conflict, which ensures that the same source model elements are not handled by more than one transformation rule in the MTr.
- Rule injectivity, which ensures that each target model element is generated by exactly one transformation rule in the MTr.

Existing MTr verification approaches express these transformation contracts in two ways:

- In a contract language which does not consider the formalism used during the verification process. The most popular contract language is OCL [3, 21–25, 47, 72, 75–78, 105]. An alternative is a visual contract language such as that used in the DSLTrans approach [87].

- In a formalism that is also used in the verification, e.g. FOL [49, 63, 98, 99], CPN [56, 120].

2.3.4 LRQ4: To What Extent are the Existing Approaches Supported by a Tool?

To answer this question, we extract information about the tool (if provided) used by each reference, including the tool's name, its input, the reasoning engine, the soundness, and the level of automation (in the sense that user intervention is not required to steer the verification process). These tools are categorised by the formal methods they use.

Simulation

Wimmer et al. have designed the TROPIC tool to encode a Query/View/Transformation relational (QVTr) MTr into CPN [120]. It allows existing CPN execution engines to automatically simulate QVTr transformations against a specific source model. The simulation is checked for various CPN properties, where each property implies certain criteria of the given model transformation.

Guerra and Lara translate a QVTr model transformation and EMF models into CPN using their CPNTools system [56]. Compared to the work of Wimmer et al., the principal differences are the encoding of certain QVTr concepts (e.g. check-before-enforce semantics, model matching and check-only scenarios of QVTr). Moreover, their encoding allows more simple Petri nets but increases the encoding complexity in certain circumstances. For instance, TROPIC encodes classes and attributes in separate places, which produces more complex CPN. However, it avoids data duplication when handling metamodels with inheritance.

Troya and Vallecillo give a detailed operational semantics for the ATL language in terms of rewriting logic using the Maude system [113]. The models and metamodels are encoded using different user-defined terms. Each ATL transformation rule is encoded as a rewriting rule to indicate how the state of model transformation is updated (where the state consists of source and target models and traces between them). The goal is to produce an alternative implementation of ATL in Maude, which allows for automatic input-driven simulation and reachability analysis of a given ATL transformation.

Model Checking and Model Finding

DSLTrans is a simple visual language with primitive constructs [9]. The transformation rules in DSLTrans are organised into layers, and any constructs which imply unbounded recursion or non-determinism are avoided. Due to this reduction in expressiveness, Lücio et al. argue that the confluence and termination of designed model transformations can be guaranteed by construction [87]. Therefore, they enumerate the state space of a given DSLTrans model transformation and represent it as a tree, where each node corresponds to a possible execution of transformation rules for an equivalent class of inputs. They formally prove that the state space of DSLTrans model transformations is always finite (which implies termination). Such finiteness is the key to designing a practical model checker for DSLTrans. As a proof of concept, Lücio et al. also develop an off-the-shelf model checker in SWI-Prolog to automatically check the syntactic correctness of target models by walking through the state space of a DSLTrans model transformation [87, 88]. This model checker is extended by Selim et al. to check commonly occurring properties, e.g., multiplicity invariants [106].

Anastasakis et al. have designed the UML2Alloy tool to perform automatic model finding [3]. The novelty of their work is in the use of Alloy, which is a verification language for SAT-based model finding [62]. Anastasakis et al. use Alloy as an IVL to ease (i) the encoding of metamodels (enriched with syntactic correctness contracts expressed in OCL) and MTr to Alloy; (ii) the generation of SAT formulas from Alloy. Anastasakis et al. also demonstrate how to use UML2Alloy to check whether the model transformation can produce well-formed target models. The case study is described in a general form and is not specific to any particular model transformation language. Later, Bütter et al. demonstrate the UML2Alloy tool for a declarative subset of the ATL model transformation language [22]. They show how to verify an ATL model transformation against syntactic and semantic correctness contracts written in OCL. However, the main problem in UML2Alloy is that Alloy does not naturally support numeric constraints. Thus, it has poor performance when solving arithmetic constraints [122].

The USE validator is another model finding tool that is based on SAT-solving. It supports a larger subset of OCL compared to UML2Alloy (e.g. multiple inheritance, OCL sequence and bag datatypes) [71]. It has been used to check model transformation refinement [23], verify model transformation in the automotive domain [105], and verify legitimate delegations in security-critical systems [47].

Jackson et al. have designed the Formula framework for SMT-based model finding [63]. The framework is based on the Z3 SMT solver [44]. The main contribution is that they systematically encode metamodels and MTr using algebraic data types. The contracts are given by FOL. Consequently, they can use their framework to find models that witness violations of syntactic correctness in the given MTr.

UML2CSP is a standalone Java application capable of verifying metamodels with OCL constraints. It is not specifically designed for MTr verification. However, Cabot et al. propose a method to derive OCL invariants from QVTr MTr in order to enable their verification and analysis in UML2CSP [24]. Their approach is also applicable to verify Triple Graph Grammar (TGG)-based graph transformations, which shows its generality. Cabot et al. send a transformation model and OCL transformation contracts to UML2CSP. Here, the transformation model refers to a unified structural description of the source metamodel, the target metamodel, and the relationship between them that is established by a model transformation [15]. Then, UML2CSP allows the user to set the search space, and select the transformation contracts to be verified. Finally, it runs the CSP solver to automatically find a legal model instance without user intervention. If the verification process succeeds, the output of UML2CSP is a graphical representation (i.e. a UML object diagram) of the found model instance as a proof. Otherwise, no output is provided.

Stevens presents a novel usage of game theory to verify the semantic correctness of QVTr [111]. The idea is that the verification tool consists of a verifier and a refuter. The verifier confirms the semantic correctness of QVTr transformations, whereas the refuter's objective is to disprove it. Then, the semantics of each QVTr transformation is encoded into the verification tool to determine how the game should progress. Finally, the verification of its semantic correctness succeeds if the verification tool finds a winning strategy for the verifier, and fails otherwise. However, no details are given about how to implement such a verification tool.

Theorem Proving

Combemale et al. present a pen and paper bi-simulation proof to show that the ATL MTr generates a Petri nets model that preserves the operational semantics of an executable SPEM model [35]. The goal of their work is to define a translational semantics for the source metamodel, i.e. defining the behavioural semantics of the source metamodel by translating to a target metamodel that has an existing formally defined semantics. This would enable reliable verification of source domain properties on the target domain.

Ledang and Dubois formalise metamodels and declarative transformation rules into the B formalism [79]. They perform model transformation verification on a specific formalised source model. The corresponding target model is incrementally built with respect to the transformation rule formalisation. Their focus is to interactively prove semantics preservation (encoded as invariants) while building the target model.

Calegari et al. encode the ATL MTr and its metamodels into inductive types [25]. The contracts for semantic correctness are given by OCL, which are translated into logical predicates. As a result, they can use the Coq proof assistant to interactively verify that the MTr is able to produce target models that satisfy the given contracts.

Inspired by the proof-as-program methodology [60], there is a line of research which develops the concept of proof-as-model-transformation methodology [28, 49, 72, 98, 99]. At its simplest, the idea is to represent the metamodels as terms. Then, each MTr and its contracts are encoded together as a $\forall\exists$ type. Type theory (for the lambda-calculus) provides the basis of a proof system to verify the encoded $\forall\exists$ type. Finally, a model transformation can be extracted from the proof.

The proof-as-model-transformation methodology is first seen in Chan's work [28]. However, few details are given regarding the kind of properties verified. Later, Poernomo outlines a method for representing metamodels and models as types. He follows the classical approach in type theory to formally specify each model transformation rule as a $\forall\exists$ type [98]:

```
∀ x: PIL • I(x) ⟹ (∃ y: PSL • O(x, y))
```

Specifically, (i) PIL and PSL are source and target metamodel types, and (ii) $I(x)$ specifies a pre-condition on the input model x for the transformation to be applied, and (iii) $O(x,y)$ specifies required properties of the output model y. Moreover, each $\forall\exists$ type is then proven using the typing rules of PVS to obtain an inhabiting term [108], where a function program can be extracted from the inhabiting term to represent the model transformation.

However, Poernomo's work does not consider the scheduling of transformation rules. This is addressed by Poernomo and Terrell using partial order contracts to link rules together [99]. The approach is similar to the previous work of Poernomo himself. However, the transformation is given as a series of mapping rules, defined over metamodels via a partially ordered traversal of the source metamodel from an given initial root element:

```
∀ x: PIL • I(x) ⟹ (∃ y: PSL • O(x, y) ∧ traverses(x))
```

Notice the introduced *traverses(x)* predicate stands for the sub-transformations of source elements that are below the root element x under a pre-defined partial ordering. Then, the proofs are carried out interactively in the Coq theorem prover.

Fernández and Terrell extend Poernomo and Terrell's work [49]. They show how to assemble the proof of a potentially large totally ordered model transformation, by decomposing it into a number of smaller proofs which are easier to prove. In addition, the metamodels are defined as Coq co-inductive types to allow bi-directionality and circular references.

The Reactive System Development Support of UML (UML-RSDS) is a tool-set for developing correct model transformations by construction [73, 78]. The UML-RSDS is first introduced by Lano in 2006 [72]. It uses a combination of UML and OCL to create a model transformation design, instead of using types. UML use-case diagrams and activity diagrams are used to graphically create a MTr. The OCL contracts can be used to constrain the source and target metamodels. Then, the MTr is verified against its contracts by translating both into the B Abstract Machine Notation. Finally, the verified model transformation design can be synthesised to an executable transformation implementation (such as a Java program or an ATL transformation). Developing on the UML-RSDS, Lano et al. further apply design patterns to model transformation to design modular and verifiable model transformations [76, 78]. As a result, the model transformation constructed using design patterns produces its model transformation implementation in a straightforward manner and can be more easily verified against model transformation contracts. Examples of these transformations are demonstrated by multiple model transformation designs written in UML-RSDS (e.g. model migration of activity diagrams, UML to relational schema and state machine slicing) [75, 76].

Finally, Büttner et al. automate the process of theorem proving by a novel use of SMT solvers [21]. The built-in background theories of SMT solvers give enhanced expressiveness to handle constraints over an infinite domain. Specifically, Büttner et al. translate a declarative subset of the ATL and OCL contracts (for syntactic and semantic correctness) directly into FOL formulas. These formulas represent the execution semantics of the ATL transformation, and are sent to the Z3 SMT solver to be discharged. The result implies the partial correctness of an ATL transformation in terms of the given OCL contracts. Later, Lano et al. also adapt the Z3 SMT solver to automate theorem proving within UML-RSDS [77].

The answers to our literature review questions are summarised in Table 2.1. The first column shows the related references (the references in the same line indicates that they apply the same tool to perform the MTr verification). The second column indicates the formal method applied (where SI for simulation, MC for model checking, MF for model finding and TP for theorem proving). The third and fourth column shows respectively the formalisation applied and type of contracts verified (Section 2.3.3) by each approach. The remaining columns characterise the tool (if any) provided by each approach.

Ref	FM	Formalisation	Contracts Type	Tool						
				Name	Input			Engine	Sound	Auto.
					Meta-/model	Contracts	MTr			
[120]	SI	CPN	C1,C4,C5,C6	TROPIC	MOF	CPN	QVTr	TROPIC	N/S	Y
[56]	SI	CPN	C1,C4,C5,C6	CPNTools	EMF	CPN	QVTr	CPNTools	N/S	Y
[113]	SI	Rewriting Logic	C4,C6	N/S	MOF	N/A	ATL	Maude	N/S	Y
[9, 87, 88, 106]	MC	Graph Theory	C1	DSLTRANS	EMF	Graph Pattern	DSLTrans	DSLTRANS	N/S	Y
[3, 22]	MF	Relational Logic	C1,C2	UML2Alloy	MOF	OCL	QVTr,ATL	Alloy	N/S	Y
[23, 47, 105]	MF	FOL	C1,C2	USE	MOF	OCL	ATL	SAT	N/S	Y
[63]	MF	Algebra	C1	Formula	MOF	FOL	General	Z3	N/S	Y
[24]	MF	FOL	C1	UML2CSP	MOF	OCL	QVTr	CSP	N/S	Y
[111]	MF	FOL	C2	N/S	General	FOL	QVTr	N/S	N/S	N/S
[35]	TP	FOL	C2	N/A	MOF	FOL	ATL	Manual	N/S	N
[79]	TP	Set Theory	C1,C2	N/S	MOF	FOL	General	B	N/S	N
[25]	TP	Type Theory	C1	N/S	MOF	OCL	ATL	Coq	N/S	N
[28]	TP	Type Theory	N/S	N/S	MOF	N/S	General	N/S	N/S	N
[49, 98, 99]	TP	Type Theory	C1,C2	N/S	MOF	FOL	General	Coq	N/S	N
[21]	TP	FOL	C1,C2	N/A	MOF	OCL	ATL	Z3	N/S	Y
[72–78]	TP	FOL	C1,C2	UML-RSDS	UML	OCL	ATL	AMN-B,Z3	N/S	Semi

Table 2.1 Literature Review Summary

2.4 Conclusion from our Literature Review

In this section, we discuss a number of findings developed from doing this literature review that, in our opinion, are worth noticing.

2.4.1 Bounded and Unbounded Verification

Essentially, simulation, model checking and model finding approaches are bounded. This means the MTr will be verified against its contracts within a given search space (i.e. using finite ranges for the number of models, associations and attribute values). Bounded approaches are usually automatic, but no conclusion can be drawn outside the search space. However, the result of bounded verification is a strong indication of MTr correctness if wide enough bounds are chosen for the search.

Theorem proving approaches are unbounded. This is preferable when the user requires that contracts hold for MTr over an infinite domain. However, most of the theorem proving approaches require guidance and expertise from the user [25, 28, 35, 78, 98, 99]. This can be ameliorated by a novel use of SMT-solvers such as that presented by Büttner et al [21].

2.4.2 Lack of an Intermediate Verification Language

The most common way to implement MTr verification tools is using a 2-phase design. The first phase is to formalise the source and target metamodels, the model transformation to be verified, the transformation contracts (if any) into a mathematical model (model-based verification) or the chosen formalism (proof-based verification). The second phase is to reason about the generated mathematical model or formula. This is usually conducted with the help of solvers or tools that are specialised for reasoning in the chosen formalism, or performed in a theoretical way using pen and paper.

We conjecture that the missing piece in this 2-phase design is an IVL that bridges between the front-end MTr language and the back-end solver/tool. It allows the designer focus on encoding verification tasks for the front-end MTr language, and delegates the communication with the back-end solver/tool to the IVL. In addition, various encodings (e.g. semantics of transformation contracts, metamodels) can be encapsulated as modules or libraries. These modules or libraries can be reused when designing verification tools for different MTr languages. In conclusion, we believe the reusability and modularity of the designed verification tool can be enhanced by using an IVL.

To our knowledge, UML2Alloy is the only verification tool that draws on an IVL. In its design, Alloy has been used as an IVL for SAT-based model finding [3]. However, the IVL has not yet been adopted in theorem proving approaches for MTr.

2.4.3 Choosing an Intermediate Verification Language

Boogie is a procedure-oriented IVL based on Hoare-logic [7]. The Boogie programs are verified by the Boogie verifier. The Boogie verifier uses an efficient SMT solver, i.e. the Z3 theorem prover, at its back-end [44]. If a Boogie program is not verified, the Boogie verifier will represent the result from Z3 as program traces, to help locate the error in the Boogie program.

A Boogie program consists of imperative constructs (e.g. variables, expressions, procedures, or procedure implementations), which we believe are suitable to formulate the proof obligations for the correctness of MTr. It also consists of mathematical constructs (e.g. type, constant, function and axiom declarations), which we believe are suitable to modularise the design of verification tool.

At the time of this thesis being written, translations into Boogie exist for several languages, including C# [8], C [39], Dafny [82], Java [80], and Eiffel [114], which show its applicability.

Another widely used IVL is Why3 [52]. Both Boogie and Why3 are based on FOL with polymorphic types, and have mature implementations to parse, type-check, and analyse programs. Verifier designs that currently target Why3 exist for Java [90], C [36], Ada [57], and B [45].

Modularisation of Why3 is introduced by the concept of a **theory**. A theory is a unit to organise related declarations (functions, constants, axioms, lemmas and etc.), e.g. the theory of sets. It can be imported by a Why3 program or by another theory to reuse its declarations. At the time of this thesis being written, Why3 already has a rich set of built-in theories.

An advantage of Why3 is that it is able to target multiple theorem provers [51]. One benefit is that it is able to assign a specific verification task to the suitable theorem prover. For example, for arithmetic problems, CVC4 is more efficient than other theorem provers; whereas Vampire is more efficient for first-order formulas [27]. Another benefit is to enable cross verification to enhance the credibility of verification results. The only side-effect we can think of is when the theorem provers do not agree on the verification result:

- It could be due to the deficiency of a theorem prover for a specific verification task. For example, choosing a SAT solver for SMT-oriented tasks.

- It could also be due to the unsound translation from Why3 to the target theorem prover. For example, due to the lack of documentation for the Yices theorem prover, the theory of Euclidean division (on negative numbers) was erroneously translated in the early version of Why3 [55].

However, the Why3 users can always choose a set of theorem provers to prove specific verification tasks, or choose to define a high success criteria for verification (e.g. the verification must succeed on all the targeted theorem provers).

We concentrate on Boogie in this research, as Boogie was the IVL that we were most familiar with when we commenced our research. However, we believe all results can be reproduced in Why3 (a coarse roadmap is demonstrated in Appendix E) or other IVLs with comparable functionality.

2.4.4 Transformation Contracts

Another observation is that OCL is one of the most popular languages for expressing the transformation contracts. This could be because the design of OCL is intrinsic to navigate/-manipulate metamodels and models, and it is widely accepted by the MDE community.

However, OCL can sometimes be verbose, and difficult to read/write [116]. Moreover, the verifier designer needs to decide how to formalise the semantics of OCL into the reasoning engine. This adds an extra complexity to the verifier implementation.

Thus, some of the existing approaches chose the formalism that is understood by the reasoning engine to directly express the transformation contracts, e.g. FOL [49, 63, 98, 99]. However, as OCL is widely accepted by the MDE community, using another formalism to express contracts becomes counter-intuitive, and could increase the learning curve when working with MTr verification.

We are convinced that an IVL can provide a flexible solution to express transformation contracts. First, the semantics of OCL can be built into an IVL as libraries in an incremental and modular way. These libraries can be reused to ease the development of OCL semantics in the reasoning engine. These libraries can also be imported on demand to avoid semantic differences among OCL formalisations (e.g. null value of OCL). Second, since most IVLs have an a easy-to-learn syntax and semantics, the users can also work directly with an IVL to express transformation contracts if they prefer.

2.4.5 Verifier Soundness

All the approaches we found rely on deriving a formula/mathematical model to represent the execution semantics of MTr. However, the reliability of this derivation has not been considered [1]. If the derivation incorrectly represents the execution semantics of the MTr, then the soundness of the verifier is compromised. It means the user of a verified MTr could experience unexpected runtime behaviour even if the verifier concludes that the MTr met its expectation.

Translation validation is a technique used in compiler verification to ensure each individual compilation is followed by a validation phase. The validation phase verifies that the compiled program produced on each run correctly implements the source program [97]. Its essential idea is to find a semantic framework that can faithfully represent both the source and compiled program. Then, a formalisation of the notion of "correct implementation" needs to be defined. Finally, a proof needs to be developed to show the compiled program correctly implements the given source program.

For MTr, recall that to be executable, the transformation is usually compiled to low-level bytecode. We believe it is feasible to adopt the translation validation approach to ensure the soundness of the verifier, i.e. to be able to check consistency between the execution semantics of each transformation and the runtime behaviour of its corresponding bytecode. However, we also anticipate that certain changes will need to be made while adopting the translation validation approach. This is mainly because of the domain-specific properties that only reside in the compilation of MTr, e.g. the bytecode instructions that are specifically designed for model handling.

2.5 Summary

In conclusion, through analysing the literature, we choose theorem proving over bounded techniques to allow deductive formal verification of MTr that quantifies over an infinite domain. Most IVLs have an easy-to-learn syntax and semantics. They bridge between the front-end MTr language and the back-end solver, and are capable of encapsulating components of the verification tool as modules or libraries. Thus, we are convinced that using an IVL is the most suitable approach to systematically designing modular and reusable verifiers for the given MTr language. The same reasons convince us that using an IVL can contribute to flexibly express transformation contracts. In this research, the IVL we choose to concentrate on is Boogie. Finally, the soundness of the designed verifier is very important. It shares similarities to compiler verification. Thus, we suggest adapting the translation validation approach from compiler verification to ensure the soundness of the designed verifier.

Chapter 3

A Modular Semantics for EMF Metamodels and OCL in Boogie

In this chapter, we describe a semantics for EMF metamodels (Section 3.2), and a semantics for OCL (Section 3.3). They form the main results of this chapter. We encode both of them in the Boogie IVL as libraries (Section 3.1). These libraries can be reused across different verifier designs for MTr. This chapter concludes with a discussion of the consistency and completeness issues for our encoded Boogie libraries (Section 3.4).

3.1 Introduction to the Boogie Intermediate Verification Language

A standard technique in theorem-prover-based program verification is to transform a given program and its proof obligations (i.e. what conditions need to hold for the program to be considered correct) into a set of logical formulas (a.k.a VCs) whose validity implies the program correctness. The VCs are then processed by the theorem prover, where a successful proof implies the correctness of the program (with respect to its proof obligations), and a failed proof may give an indication of a possible error in the program.

Previous automatic program verifier designs suggested that the complex task of generating VCs for high-level programming languages can be managed by separating the task into two steps [7, 50, 82]: a transformation from the program and its proof obligations into a program written in an IVL, and then a transformation from the IVL program into logical formulas.

The IVL bridges between the front-end high level programming language and the back-end theorem prover. The benefit is to focus on generating proof obligations that prescribes what correctness means for the front-end language in a structural way, and to delegate the task of interacting with theorem provers to the IVL.

Boogie is a procedure-oriented IVL based on Hoare-logic [7]. The verification of Boogie programs is performed by the Boogie verifier which uses the Z3 theorem prover at its back-end [44]. If the Boogie program is not verified, the Boogie verifier will represent the result from Z3 as program traces, to help locate the error in the Boogie program. At the time of this thesis being written, translations into Boogie exist for several languages, including C# [8], C [39], Dafny [82], Java [80], and Eiffel [114], which shows its applicability.

A Boogie program consists of declarations for types, functions, constants, axioms, expressions, variables, procedures, or procedure implementations.

Imperative statements (such as assignment, if and while statements) are provided by Boogie to structure the procedure implementations. FOL contracts (i.e. pre/postconditions expressed by Boogie expressions) are supported to specify procedures. A Boogie program is considered verified if its procedure implementations satisfy their corresponding contracts.

```
1  procedure Mc91(n: int) returns (r: int);
2    ensures 100 < n ⟹ r = n − 10;
3    ensures n ≤ 100 ⟹ r = 91;
4
5  implementation Mc91(n: int) returns (r: int)
6  { if (100 < n) {
7      r := n − 10;}
8    else {
9      call r := Mc91(n + 11);
10     call r := Mc91(r);}
11 }
```

Fig. 3.1 Boogie encoding of the McCarthy-91 function

To demonstrate what a Boogie program looks like, we show the Boogie encoding of the McCarthy 91 function in Fig. 3.1 [89]:

- First, the signature of the Boogie procedure specifies that the McCarthy-91 function takes one input *n* and one output *r* of type *int* (line 1).

- Then, the postconditions that establish the relationship between the input and output are specified by two *ensures* clause (line 2 - 3). That is if the input is greater than 100, return the input minus 10 as the output; otherwise always return 91.

- The procedure implementation uses a Boogie *if* statement to form a case distinction according to the input value (line 5 - 11). That is if the input value is greater than 100, the output is assigned the input minus 10. Otherwise, two recursive calls are invoked sequentially to compute the output.

Although the formal proof of McCarthy 91 can be done manually, the Boogie verifier can prove it automatically by applying basic principles for verifying recursive calls (e.g. inlining the contracts of the recursive call).

In addition, Boogie allows type, constant, function and axiom declarations, which are mainly used to encode libraries that define data structures, background theories and language properties. These features are used in this chapter to encode Boogie libraries that define semantics for EMF Metamodels and OCL.

Notice that we do not intend to give a full semantics of the Boogie IVL in this chapter, since a comprehensive manual for Boogie has been presented by Leino [81].

3.2 Library for the Semantics of EMF Metamodels

Metamodelling concepts share many similarities with OO programming language constructs. Thus, the Boogie encoding of OO programs can be adapted to encode the semantics of EMF metamodels. However, because of the subtle semantic differences between metamodelling concepts and OO constructs, such adaptation requires certain customisations. These customisations are the focus of this section.

The abstract syntax of supported features of the EMF metamodels is shown in Fig. 3.2. Each classifier should belong to a metamodel and must have a name. Optionally, it can have a list of structural features. It could be inherited from other classifiers. However, cyclic inheritance is not allowed [63]. Moreover, declaring a classifier to be abstract or an interface is allowed. A structural feature (also known as a member field) of a classifier must have a name. It is declared to be of a given type, i.e. type of bool, int, string, or a reference to (the elements of) a classifier. A structural feature is called an *attribute* if its type is bool, int or string. It is called an *association* if its type is a reference type. In addition, the multiplicity can be specified to indicate the cardinality of the structural feature. It is denoted by *lower..upper*, where *lower* and *upper* are the bounds of the multiplicity.

In the following sections, we introduce the semantics of supported features of the EMF metamodels. The semantics of these features are given by their corresponding Boogie encoding.

⟨*Metamodel*⟩ ::= MM(name: string)

⟨*Classifier*⟩ ::= Clazz(mm: ⟨*Metamodel*⟩, name: string, fields: seq⟨*StructuralFeature*⟩,
parents: seq⟨*Classifier*⟩, isAbstract: bool, isInterface: bool)

⟨*type*⟩ ::= bool | int | string | ref(cl: ⟨*Classifier*⟩)

⟨*StructuralFeature*⟩ ::= Field(owner: ⟨*Classifier*⟩, id: string, ty: ⟨*type*⟩,
lower: int, upper: int)

Fig. 3.2 Abstract syntax of the supported features of EMF metamodels

3.2.1 Semantics of EMF Classifiers

Our library for the semantics of the EMF metamodels declares a non-primitive Boogie type *ClassName* to type each constant that represents a classifier name:

```
type ClassName;
```

Example 3.2.1. The classifiers *Entity* and *Relship* from the *ER* metamodel (shown in Appendix B.1) can be declared in Boogie as follows:

```
const unique ER$Entity: ClassName;
const unique ER$Relship: ClassName;
```

Some explanation is in order. First, each classifier is prepended with its metamodel name to avoid name conflicts between source and target metamodels, and followed by a separator symbol $ for readability. Second, each constant is declared with the *unique* modifier that shows the constant has a value that is different from the values of other unique constants of the same *ClassName* type.

An inheritance relationship can be defined via a partial order relationship (i.e. reflexive, transitive, and antisymmetric relationship) between a classifier and a set of parent classifiers (the set can be empty).

Example 3.2.2. Provided that there are two classifiers *Entity* and its sub-classifier *SpecialEntity*, their declarations in Boogie can be encoded as follows:

```
const unique ER$Entity: ClassName extends ;
const unique ER$SpecialEntity: ClassName extends ER$Entity;
```

The first line simply conveys that *Entity* does not have any parents. The second line conveys that *SpecialEntity* inherits from *Entity*.

Multiple-inheritance is allowed. Its encoding can be tricky since *immediate parent* modifiers can be involved among constants. Being an immediate parent *x* of constant *y* means that there does not exist any other constants *any* such that *x extends any and any extends y*.

The following example is a violation of the immediate parent declaration, which introduces *Entity* in the middle of *SpecialEntity* and *NamedElement*, and should be avoided in encoding.

```
const unique NamedElement: ClassName extends ;
const unique ER$Entity: ClassName extends NamedElement;
const unique ER$SpecialEntity: ClassName extends ER$Entity , NamedElement;
```

Finally, a subtle but important encoding is to use a *complete* modifier on every classifier. This is to convey that the complete set of classifiers that extends from a given classifier is known. This is quite useful when quantifying over an abstract classifier, when the quantification needs to know exactly which sub-classifiers are inherited from an abstract classifier. We have not seen any OO verifier with a mature implementation which enforces such constraints on the classes. The reason could be that in the context of OO programming, the complete set of classes that extends from a class is generally not yet fixed during program development.

Example 3.2.3. We show the complete classifier encoding of the *ER* metamodel in Fig. 3.3.

```
const unique ER$Entity: ClassName extends complete;
const unique ER$Relship: ClassName extends complete;
const unique ER$ERSchema: ClassName extends complete;
const unique ER$ERAttribute: ClassName extends complete;
const unique ER$RelshipEnd: ClassName extends complete;
```

Fig. 3.3 Boogie encoding of classifiers in ER metamodel

3.2.2 Semantics of EMF Structural Features

To facilitate typing the structural features in the library for the semantics of EMF metamodels, *bool* and *int* types are primitively supported in Boogie, and the *string* type is modelled as a sequence of *int*. A nullary type constructor *ref* models the reference type:

```
type ref;
```

To distinguish between associations of different classifiers, a *dtype* function that maps each association to its allocated classifier is declared:

```
function dtype(ref) returns (ClassName);
```

Different multiplicities on an association are distinguished by the result of evaluating the result of their *dtype* function:

- An association with upper bound 1 will return its allocated classifier.
- An association with upper bound * will always return the classifier *System.array*.

Most OO languages support the concept of multiplicity indirectly, e.g. Java, C#. Thus, the concept of multiplicity is usually not reflected in their verifier design. However, multiplicity is commonly used in MDE, which makes its encoding important for MTr verification to work in practice.

In the library for the semantics of the EMF metamodels, a unary type constructor is used to type the structural features:

```
type Field α;
```

Example 3.2.4. The classifier *ERSchema* from an *ER* metamodel with the structural features *name*, *relships* and *entities* is declared as follows:

```
const unique ER$ERSchema.name: Field String;
const unique ER$ERSchema.relships: Field ref;
const unique ER$ERSchema.entities: Field ref;
```

Some explanation is in order. First, each structural feature is mapped to a unique constant of type *Field* α, where α is of primitive type (i.e. *int*, *bool* and *string*) for each attribute, and is of *ref* type for each association. Second, all the constants generated for attributes or associations are extended with the corresponding classifier name to ensure their uniqueness.

3.2.3 Burstall-Bornat Memory Model

The Burstall-Bornat memory model is commonly used in OO verifier design to represent the runtime heap [20]. Its general idea is to use an updatable array *heap* to organise the relationships between runtime objects. Such an array is defined with the following Boogie type:

```
type HeapType = <α>[ref, Field α] α
```

The *HeapType* is a type synonym to abbreviate the map type that is defined on its right hand side. Such a two-dimensional map type is defined in a polymorphic manner (i.e. parametrised by the bound type identifier α). It allows the mapping of memory locations (identified by a runtime object and a field) to values of type α.

In our library for the semantics of the EMF metamodels, the Burstall-Bornat memory model is used to organise the relationships between runtime elements of classifiers, which allows the mapping of memory locations (identified by an element of a classifier, and a structural feature) to values.

The domain of the *heap* includes allocated as well as unallocated elements. To distinguish between these two, it is useful to add a structural feature *alloc* of type *Field bool* and set it to true when an element is allocated. To ensure safe memory access, certain operations should only operate on the allocated elements.

A memory access expression *o.f* is now seen as the expression *read(heap,o,f)*. An assignment *o.f:=x* is understood as the expression *update(heap,o,f,x)*, i.e. changing the value of the heap at the position given by the element *o* and structural feature *f*, to the value of *x*.

There are several advantages of adopting the Burstall-Bornat memory model to express the semantics of the EMF metamodels:

- One is that it organises runtime model elements in a single updatable array, which can be flexibly passed as an argument to the places where it is needed (e.g. as in the *read* and *update* function) [81].
- Another advantage is that it allows quantification over fields [81]. This is convenient when expressing the semantics of ATL (see the frame problem in Section 4.2.1).
- A further advantage is the enhancement of verifier interoperability. Verifiers that are built using the same IVL and share the same memory model form a verification ecosystem, which allows information to seamlessly flow between them.

3.3 Library for the Semantics of OCL

A major advantage of adopting Boogie for verifier design is reusability. That is the verifier designers can draw on the existing Boogie libraries to implement their verifiers, and then the libraries that they develop can be made available to others.

This is the case while developing our library for the semantics of OCL. In particular, we encode a subset of OCL data types that are supported in ATL, i.e. OCLType, OCLAny, Primitive (*OCLBool*, *OCLInteger*, *OCLString*) and Collection (*OCLSet*, *OCLOrderedSet*, *OCLSequence*, *OCLBag*). Overall, 78 OCL operations are supported on the chosen data types. This encoding is based on a Boogie library for the theory of *set*, *sequence* and *bag* provided by the Dafny verifier [82]. Twenty-three Boogie functions from this library are directly reused in our encoding. On top of these, we further introduce the theory for *OCLOrderedSet* collection data type (with 3 OCL operations) in Boogie, and 8 OCL iterators on *OCLSequence* and *OCLOrderedSet* data types (i.e. exists, forall, isUnique, one, any, select, collect and reject iterators).

3.3.1 Semantics of OCLType

The data type *OCLType* corresponds to the definition of each type instance specified by OCL. For example, *ERSchema* from the *ER* metamodel is an *OCLType*. It corresponds to the Boogie type *ClassName*.

Each *OCLType* is associated with an *allInstances* operation (written as *operand.allInstances()*), which returns a list of the currently allocated instances whose classifier is the **kind** of the specified *OCLType*. The *allInstances* operation is encoded into Boogie by the following function:

```
function OCLType#allInstance(heap: HeapType, cl: ClassName): Seq ref;
  axiom (∀ heap: HeapType, cl: ClassName, r: ref •
    Seq#Contains(OCLType#allInstance(heap, cl), r) ⟺
      r ≠ null ∧ read(heap, r, alloc) ∧ dtype(r)<: cl);
```

The semantics of Boogie functions are usually given as axioms. The axioms are expressed using Boogie expressions (i.e. FOL expressions such as variables, arithmetic and equality as well as ordering relations, boolean connectives, simple arithmetic operators, logical quantifiers and a pre-state operator). For example, the axiom indicated here is the main axiom for the *allInstances* operation. It specifies that the function returns a sequence of *ref* where each *ref* is a non-null reference, and has to be allocated on the given *heap*, and is a subtype of *cl*.

It is possible to write inconsistent axioms in Boogie (e.g. "axiom false;"), which trivially renders any proof obligations valid. Therefore, it is the verifier designers' responsibility to make sure that all the declared axioms are consistent. We will further discuss the state of the art that prevents the inconsistent axiomatic system in Section 3.4.

3.3.2 Semantics of OCLAny

OCLAny behaves as a super-type for all the data types. It does not correspond to any Boogie constructs. OCL supports a set of operations for *OCLAny* that are common to all existing data types (written as operand.operation_name(parameters)). In our library for the semantics of OCL, the supported operations are:

- comparison operators: =, <>;
- OCLIsUndefined() returns a boolean value stating whether the operation's operand is undefined;
- OCLIsKindOf(t : OCLType) returns a boolean value stating whether the operation's operand is either an instance of *t* or of one of its subtypes;
- OCLIsTypeOf(t : OCLType) returns a boolean value stating whether the operation's operand is an instance of *t*.
- OCLType() returns the *OCLType* of operation's operand;

The *polymorphic* function is used to encode these operations on *OCLAny*. The signature of a polymorphic function is parametrised by type identifier(s), which are replaced by the concrete data type(s) while axiomatizing the function.

Example 3.3.1. The *OCLIsUndefined* operation is encoded as follows:

```
function OCLAny#isUndefined<α>(h: HeapType, elem: α): bool;
  axiom (∀ h: HeapType, i: int • ¬OCLAny#isUndefined(h,i));
  axiom (∀ h: HeapType, b: bool • ¬OCLAny#isUndefined(h,b));
  axiom (∀ h: HeapType, s: string • ¬OCLAny#isUndefined(h,s));
  axiom (∀ h: HeapType, r: ref • OCLAny#isUndefined(h,r) ⟺ (r=null ∨ ¬read(h,r,alloc)));
```

Some explanation is in order. First, all of our polymorphic functions are axiomatised over certain data types (i.e. *bool*, *int*, *string* and *ref*). Axiomatisation for other data types needs to be introduced on demand by the user of the verifier (which has rarely happened in our experience).

Second, one subtlety in our encoding is how to handle the two *Undefined* values for *ref* (i.e. *null* and *invalid*) [11]. The *null* value is modelled in Boogie as follows:

```
const null: ref;
```

The *invalid* value is modelled in Boogie by checking whether it is an allocated *ref* on the given heap.

3.3.3 Semantics of Primitive Data Types

Our library for the semantics of OCL supports three OCL primitive data types (*OCLBool*, *OCLInteger* and *OCLString*). The first two have native support in Boogie. The *OCLString* is mapped to an *OCLSequence* of integers. The operations supported on the primitive data types have an intuitive mapping to the Boogie constructs (Table 3.1). The first column lists the three primitive data types. The second column details the supported OCL operations on each primitive data type, whose behaviour is commented in the third column. The fourth column indicates their coarse mapping to Boogie (see Appendix A for a detailed mapping).

Data Type	**OCL Operation**	**Comment**	**Boogie**
OCLBool	and, or, implies, not	logical operators.	Default
OCLInteger	<, >, >=, <=, =, <>	comparison operators.	Default
	*, +, -, div(), mod()	binary operators.	
	abs()	unary operators.	
OCLString (where *s* is a string)	s.size()	returns the number of characters contained in *s*.	theory of *sequence*
	s.concat(s2:OCLString)	returns a string in which the string *s*2 is concatenated to the end of *s*.	
	s.substring (lower:OCLInteger, upper:OCLInteger)	returns the substring of *s* starting from the character indexed by *lower* to the character indexed by *upper*.	
	s.toUpper(), s.toLower()	respectively return an upper/lower case copy of *s*.	
	s.startsWith(s2:OCLString), s.endsWith(s2:OCLString)	respectively return a boolean value stating whether *s* starts/ends with *s*2.	

Table 3.1 Semantics of OCL primitives

3.3.4 Semantics of Collection Data Types

OCL defines four collection data types. These collection data types differ in the way that they store elements:

- *OCLSet* is a collection without duplicate elements, and its elements are not indexed;
- *OCLOrderedSet* is a collection without duplicates, and its elements are indexed;
- *OCLBag* is a collection that has duplicates, and its elements are not indexed;
- *OCLSequence* is a collection that has duplicates, and its elements are indexed;

To model these data types, the following Boogie types are introduced:

```
type Set T = [T]bool;      // OCLSet, membership of elements
type OrderedSet = Seq T; // OCLOrderedSet
type MultiSet T = [T]int;// OCLBag, no. of occurrence of elements
type Seq T;               // OCLSequence
```

Data Type	OCL Operation	Comment	Boogie
OCLSet (where *s* is a set)	s.union(s2:OCLSet), s.intersection(s2:OCLSet), s-s2	returns union/intersection/difference of two sets.	theory of *set*
	s.including(o:OCLAny), s.excluding(o:OCLAny)	returns a new set that is the same as *s* except including/excluding *o*.	
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	returns whether the object *o* is included/excluded in/from *s*.	
	s.isEmpty(), s.notEmpty()	returns whether *s* is/not empty.	
OCLBag (where *s* is a bag)	s.including(o:OCLAny), s.excluding(o:OCLAny)	return a new bag that as same as *s* except the occurrence *o* increased/decreased by 1.	theory of *multiset*
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	returns whether the object *o* is included/excluded in/from *s*.	
	s.isEmpty(), s.notEmpty()	returns whether *s* is/not empty.	

Table 3.2 Semantics of OCL collections

Data Type	**OCL Operation**	**Comment**	**Boogie**
OCLOrderedSet (where *s* is an ordered set)	s.append(o:OCLAny), s.prepend(o:OCLAny)	returns a new ordered set with the element *o* append/prepend to *s*, if *o* is not in *s*.	theory of *sequence*
	s.insertAt (n:OCLInteger, o:OCLAny)	returns a new ordered set with the element *o* added at index *n* of *s*, if *o* is not in *s*.	
	s.subOrderedSet (lower:OCLInteger, upper:OCLInteger)	returns *s* starting from the element indexed by *lower* to the element indexed by *upper*.	
	s.at(n:OCLInteger)	returns the element indexed by *n* in *s*.	
	s.first(), s.last()	returns the first/last element of *s* (OCLUndefined if *s* is empty).	
	s.size()	returns the size of *s*.	
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	returns whether the object *o* is included/excluded in/from *s*.	
	s.isEmpty(), s.notEmpty()	returns whether *s* is/not empty.	
OCLSequence (where *s* is a sequence)	s.union(s2:OCLSequence)	returns a new sequence with *s*2 appended to the end of *s*.	theory of *sequence*
	s.append(o:OCLAny), s.prepend(o:OCLAny)	returns a new sequence with the element *o* appended/prepended to *s*.	
	s.insertAt(n:OCLInteger, o:OCLAny)	returns a new sequence with the element *o* added at index *n* of *s*.	
	s.subSequence (lower:OCLInteger, upper:OCLInteger)	returns *s* starting from the element indexed by *lower* to the element indexed by *upper*.	
	s.at(n : OCLInteger)	returns the element indexed by *n* in *s*.	
	s.first(), s.last()	returns the first/last element of *s* (OCLUndefined if *s* is empty).	
	s.size()	returns the size of *s*.	
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	returns whether the object *o* is included/excluded in/from *s*.	
	s.isEmpty(), s.notEmpty()	returns whether *s* is/not empty.	

Table 3.2 Semantics of OCL collections (cont.)

The modelling of OCL collection data types is adapted from the theory of *set*, *multiset* and *sequence* provided by the Dafny verification system [82]. The operations supported on the collection data types have an intuitive mapping to the Boogie functions in its corresponding theory provided by Dafny. These are listed in Table 3.2 (structured as in Table 3.1). Again, the fourth column indicates their coarse mapping to Boogie. The detailed mappings are listed in Appendix A.

Moreover, eight collection iterators are supported (i.e. exists, forall, isUnique, one, any, select, collect and reject iterators). Notice that these iterators are only supported on the *OCLSequence* and *OCLOrderedSet* data types, since the other collection data types do not preserve order, and are thus not enumerable.

These iterators are encoded as Boogie functions and their meaning is encoded by axioms. We show the *select* iterator as an example, the encoding of other iterators is similar. The *select* iterator, written as *s->select*$(e|f)$, selects the elements that evaluate the filter expression *f* to true from the sequence *s*. Here, the iterator variable *e* refers to the current element on each iteration, which can be used in the expression *f*.

The *select* iterator is encoded in Boogie as shown in Fig. 3.4.

```
function Iterator#Select<T>(lo: int, hi: int, s: Seq T,
                            h: HeapType, f: [T, HeapType]bool): Seq T;
  // forward induction axiom, when filter expression evaluates to true.
  axiom (∀<T> lo: int, hi: int, s: Seq T, h:HeapType, f: [T, HeapType]bool •
    lo<hi ∧ f[Seq#Index(s,lo), h] ⟹
      Iterator#Select(lo,hi,s,h,f) =
        Seq#Append(Seq#Singleton(Seq#Index(s,lo)), Iterator#Select(lo+1,hi,s,h,f)));
  // forward induction axiom, when filter expression evaluates to false.
  axiom (∀<T> lo: int, hi: int, s: Seq T, h: HeapType, f: [T, HeapType]bool •
    lo<hi ∧ ¬f[Seq#Index(s,lo), h] ⟹
      Iterator#Select(lo,hi,s,h,f) = Iterator#Select(lo+1,hi,s,h,f));
  // backward induction axiom, when filter expression evaluates to true.
  axiom (∀<T> lo: int, hi: int, s: Seq T, h: HeapType, f: [T, HeapType]bool •
    lo<hi ∧ f[Seq#Index(s,hi−1), h]⟹
      Iterator#Select(lo,hi,s,h,f) =
        Seq#Append(Iterator#Select(lo,hi−1,s,h,f), Seq#Singleton(Seq#Index(s,hi − 1))));
  // backward induction axiom, when filter expression evaluates to false.
  axiom (∀<T> lo: int, hi: int, s: Seq T, h: HeapType, f: [T, HeapType]bool •
    lo<hi ∧ ¬f[Seq#Index(s,hi−1), h] ⟹
      Iterator#Select(lo,hi,s,h,f) = Iterator#Select(lo,hi−1,s,h,f));
  // splitting axiom
  axiom (∀<T> mid: int, lo: int, hi: int, s: Seq T, h: HeapType, f: [T, HeapType]bool •
    lo≤mid ∧ mid≤hi ⟹
      Iterator#Select(lo,hi,s,h,f) =
        Seq#Append(Iterator#Select(lo,mid,s,h,f),Iterator#Select(mid,hi,s,h,f)));
  // consequence axiom
  axiom (∀<T> i: int, lo: int, hi: int, s: Seq T, h: HeapType, f: [T, HeapType]bool •
    0≤i ∧ i<Seq#Length(Iterator#Select(lo,hi,s,h,f)) ⟹
      f[Seq#Index(Iterator#Select(lo,hi,s,h,f),i), h]);
```

Fig. 3.4 The Boogie encoding for the *select* iterator of OCL collection

The Boogie function for the *select* iterator is polymorphically parametrised by the bound type identifier *T*. It takes 5 parameters. The first two parameters accept the range that the iterator works on. The third parameter receives the sequence to be iterated upon. The fourth parameter takes the runtime heap (which is used to evaluate the filter expression). The last parameter accepts the filter expression.

The semantics of the *select* iterator is encoded by axioms. The first two axioms define how to select elements by increasing the range inductively (line 3 - 11). The next two axioms define how to select elements by decreasing the range inductively (line 12 - 20). Then, another axiom states that the task of sequence selection can be decomposed by splitting the input sequence into two using a pivot position *mid* (line 21 - 25). The result of selecting on the original sequence will be the same as combining the results of selecting on the split sequence. The last axiom states the fact that for every selected element the filter expression *f* evaluates to true (line 26 - 29).

Example 3.3.2. A simple *select* iterator, on a sequence of *ERAttribute*, which selects the element whose *isKey* is evaluated to *true*, can be expressed in OCL as:

```
s->select(attr:ERAttribute|attr.isKey)
```

Such an iterator yields the following call to the *Iterator#Select* Boogie function:

```
Iterator#Select(0, Seq#Length(s)-1, s, heap,
 (λ attr:ref, hp: HeapType • attr ≠ null ∧ read(hp,attr,alloc) ∧ dtype(attr)<: ER$ERAttribute
   ⟹ read(hp,attr,ER$ERAttribute$isKey)))
```

The first four arguments offer no surprise and correspond to the first four parameters of the *select* iterator. The fifth argument is a lambda expression that evaluates whether the input (that is non-null and allocated of subtype of *ERAttribute* on the given heap) has a *true* value for its *isKey* attribute.

3.4 Consistency and Completeness of Our Libraries

In this section, we discuss our libraries for the semantics of EMF metamodels and OCL in terms of their consistency and completeness.

Consistency. Inconsistent Boogie axioms inside a Boogie library would render everything trivially verified by the back-end theorem prover and would thus compromise the soundness of the verifier.

The consistency of our Boogie libraries for the semantics of metamodels and OCL are challenging theoretical problems that require well-defined and commonly accepted formal semantics for each. To our knowledge, none of these are currently available. When one becomes available, we can adapt existing techniques to reason about the consistency of our encodings. For example, one approach to show axiom consistency is to generate proof obligations for the axiomatic system [40, 41, 83]. This aims at using a SMT solver to prove the existence of each axiomatised function. A successful proof ensures the axioms of a function are consistent. The main limitation is that the generated proof obligations are too difficult to prove by the existing SMT solvers either manually or automatically. Moreover, theory interpretation approaches map from axiomatic systems to mathematical structures of a theorem prover [42, 43]. Then, the corresponding interpretation of axioms become theorems to be proved by the theorem prover. The successful proof ensures axiomatic systems are relatively consistent with the mathematical structures of the mapped theorem prover.

Two of the approaches we use while developing our Boogie libraries are "false derivable" and using test oracles. The "false derivable" approach can be used to test whether *false* is derivable from the axiomatic system [64, 91]. It is suitable for inconsistent axiomatisations that require trivial actions to reproduce. A more practical approach is to write test oracles that specify verification scenarios and their corresponding expected outcome. This approach is also used in the regression tests of the official development of Boogie to ensure its consistency on each rebuild.

Completeness. Completeness prevents false positives. There are various reasons for our Boogie libraries for the semantics of metamodels and OCL to be incomplete. One possible reason is because their Boogie encoding poses verification challenges to the theorem prover. Therefore, in future work we would like to evaluate the verification performance on different encoding of our Boogie libraries. Böhme and Moskal propose a scalable benchmark to evaluate the verification performance on different encoding of dynamic data structures (e.g. memory models) [18]. We believe their work would provide guidance on preparing benchmark and defining metrics for our evaluation.

Another reason could be that the generated VCs from our Boogie libraries are too complex to be solved by the back-end theorem prover. In order to address this problem, Leino and Rümmer illustrate the type system design of Boogie, and investigate how to translate polymorphic types and expressions into VCs that are more suitable for SMT solvers to solve [85]. Their experimental results support the idea that embedding such features in an IVL is both desirable for the implementation and performance of verifiers.

The completeness of our Boogie libraries could also suffer from the limitation of state of the art theorem provers, i.e. the undecidability of FOL when the universal quantifiers are involved [46, 53, 81]. Boogie allows triggers (a.k.a matching patterns) that determine how to instantiate universal quantifiers. The use of appropriate triggers is crucial to get good performance and desirable results from the SMT solver. Leino and Monahan describe how to use triggers effectively in the axiomatisation of summation-like comprehensions in Boogie [84], which is the guidance we followed when encoding our iterators for OCL collections. There are also other systematic approaches to infer triggers. Detlefs et al. propose an syntactical approach to automatically infer the triggers at the SMT solver level [46]. Their approach is applied in the design of the Simplify SMT solver. Ge et al. argue that such a syntactical approach is too restrictive in certain cases [53]. For example, they think it is not necessary to restrict a trigger to contain additional variables besides the bounded variables of a quantifier. Their disagreements with the Simplify SMT solver are reflected in the design of the CVC3 SMT solver.

Potential incompleteness could also be due to missing axioms in our Boogie libraries. For example, the *take* and *drop* operations of the *sequence* data type in our OCL library are encoded by just the essential axioms required to define its meaning. The auxiliary axioms such as “taking the subsequence of the original sequence, from index zero to the length of the original sequence minus one, is the same as the original sequence” are not in our encoding. Consequently, Z3 is unable to figure out the outcome of the proof obligations such as “when the subsequence takes every element of the original sequence, the first element of the original sequence is contained in the resulting subsequence”. We believe it is better to present the missing auxiliary axioms as lemmas and introduce them on demand to make the verification task smaller. Moreover, presenting only the essential axioms is our strategy that helps manual inspection and reduces the possibility of inconsistent axioms.

3.5 Summary

The main results of this chapter are two Boogie libraries for the semantics of the EMF metamodels and OCL. The goal of their development is to be able to reuse them across different verifier designs for MTr. In the next chapter, we evaluate the two Boogie libraries on the verifier design for one of the most widely used MTr languages, i.e. the ATL language, to demonstrate the feasibility of their adoption.

Chapter 4

VeriATL: a modular and reusable verifier for ATL

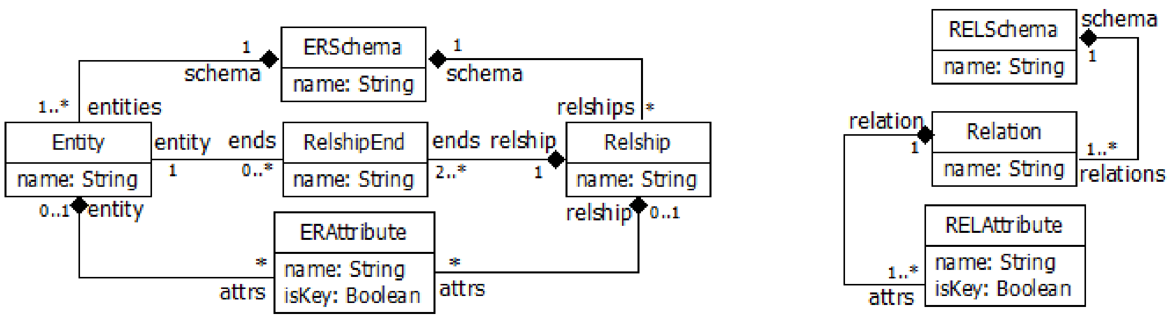
In this chapter, we introduce VeriATL, a modular and reusable verifier to ensure the correctness of the ATL transformations. Its design is based on the libraries described in Chapter 3. In Section 4.1, we present an overview of our VeriATL verifier, including the basic concepts of the ATL MTr language, and a definition for its correctness. Then, we detail the semantics of ATL (Section 4.2), which forms the main result of this chapter. The foundation of its design is based on the two modular Boogie libraries for the semantics of EMF and OCL presented in Chapter 3. We detail our implementation of VeriATL in Section 4.3. In Section 4.4, we evaluate VeriATL on two case studies, demonstrating its performance and feasibility. This chapter concludes with a discussion of the known limitations of VeriATL and a summary of the lessons we have learned (Section 4.5).

4.1 Introduction to VeriATL

ATL is one of the most widely used MTr languages in both industry and academia [66]. An ATL transformation (i.e. an ATL program) is a declarative specification that documents what the ATL transformation intends to do. The workhorses of the ATL transformation are the ATL matched rules. These rules specify the mappings between the source metamodel and the target metamodel, using OCL for both its data types and its declarative expressions. Then, the ATL transformation is compiled into an ASM implementation to be executed.

Example 4.1.1. We use the *ER2REL* transformation as the running example to demonstrate the ATL language [22]. *ER2REL* transforms the Entity-Relationship (ER) metamodel

(Fig. 4.1 (a)) into the RELational (REL) metamodel (Fig. 4.1 (b)). Both the ER schema and the relational schema have a commonly accepted semantics. Thus, it is easy to understand their metamodels.



(a) ER (b) REL

Fig. 4.1 Entity-Relationship and Relational metamodels

The *ER2REL* transformation is defined in a mapping style via a list of ATL matched rules (Fig. 4.2). The first three rules map respectively each *ERSchema* element to a *RELSchema* element (*S2S*), each *Entity* element to a *Relation* element (*E2R*), and each *Relship* element to a *Relation* element (*R2R*). The remaining three rules generate a *RELAttribute* element for each *Relation* element created in the *REL* model.

Each ATL matched rule has a *from* section where the source elements to be matched in the source model are specified. An optional OCL constraint may be added as the guard, and a rule is applicable only if the guard evaluates to true. Each rule also has a *to* section which specifies the elements to be created in the target model. The rule initialises the attribute/association of a generated target element via the binding operator (<-).

An important feature of ATL is the use of an implicit *resolve* algorithm during the target element initialisation. It is responsible for resolving the right hand side of the binding operator before assigning to the left hand side. For example, the binding *relation* <- *ent* in the *EA2A* rule on line *15* of Fig. 4.2 assigns the *Relation* element that is created for *ent* by the *R2R* rule to *relation*. The *resolve* algorithm is discussed in Section 4.2.1.

With the increasing complexity of ATL transformations, it is urgent to develop techniques and tools that prevent incorrect ATL transformations from generating faulty models. The effects of such faulty models could be unpredictably propagated into subsequent MDE steps, e.g. code generation, to produce further errors.

The correctness of an ATL transformation is defined by transformation developers using contracts. The contracts are the annotations on the ATL transformation to express the assumptions about those circumstances when it is considered to be correct. In MDE, contracts are usually expressed in OCL due to its declarative and logical nature.

```
module ER2REL; create OUT : REL from IN : ER;

rule S2S {
  from s: ER!ERSchema
  to t: REL!RELSchema (name <- s.name, relations <- s.entities, relations <- s.relships} )}

rule E2R {
  from s: ER!Entity to t: REL!Relation ( name <- s.name) }

rule R2R {
  from s: ER!Relship to t: REL!Relation ( name <- s.name) }

rule EA2A {
  from att: ER!ERAttribute, ent: ER!Entity (att.entity = ent)
  to t: REL!RELAttribute ( name <- att.name, isKey <- att.isKey, relation <- ent ) }

rule RA2A {
  from att: ER!ERAttribute, rs: ER!Relship ( att.relship = rs )
  to t: REL!RELAttribute ( name <- att.name, isKey <- att.isKey, relation <- rs ) }

rule RA2AK {
  from att: ER!ERAttribute, rse: ER!RelshipEnd
      ( att.entity = rse.entity and att.isKey = true )
  to t: REL!RELAttribute ( name <- att.name, isKey <- att.isKey, relation <- rse.relship )}
```

Fig. 4.2 ATL transformation for ER2REL model transformation

Example 4.1.2. In Fig. 4.3, two OCL contracts are given: the *unique_er_schema_names* contract (i.e. all instances of *ERSchema* have a unique name) imposed on the *ER* metamodel and the *unique_rel_schema_names* contract (i.e. all instances of *RELSchema* have a unique name) imposed on the *REL* metamodel.

```
context ER!ERSchema inv unique_er_schema_names: -- unique name of ERSchemas
  ER!ERSchema.allInstances()->forAll(s1,s2 | s1<>s2 implies s1.name<>s2.name)
----------------------------------------
context REL!RELSchema inv unique_rel_schema_names: -- unique name of RELSchemas
  REL!RELSchema.allInstances()->forall(r1,r2| r1<>r2 implies r1.name<>r2.name)
```

Fig. 4.3 OCL contracts for ER and REL

Consequently, verifying that an ATL transformation is correct with respect to the given sets of contracts can be represented as a classic Hoare-triple, i.e. assuming the contracts imposed on the source metamodel (precondition) holds, the safe execution of an ATL transformation should guarantee that the contracts are fulfilled on the generated target metamodel (postcondition).

Based on the Hoare-triple notation, a traditional approach to designing a verifier for ATL requires the encoding of the execution semantics of an ATL transformation in a formal language [22, 25, 35, 113]. Combined with a formal treatment of transformation contracts, a theorem prover can be used to verify the ATL transformation against the specified contracts. The result of the verification will imply the correctness of the ATL transformation.

VeriATL Verification System. While the traditional approach is practical from an ad-hoc point of view, there is a lack of methods for verifying the correctness of ATL transformations in an integral way, which takes the modularity and reusability of the verifier design into account. We therefore propose the VeriATL verification system (Fig. 4.4) to tackle this problem.

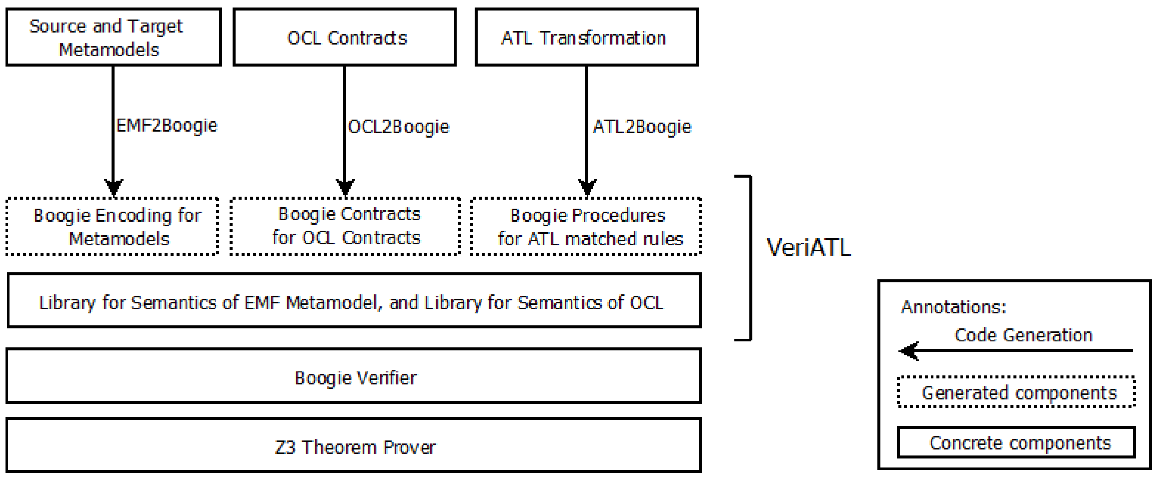

Source and Target Metamodels
OCL Contracts
ATL Transformation
EMF2Boogie
OCL2Boogie
ATL2Boogie
Boogie Encoding for Metamodels
Boogie Contracts for OCL Contracts
Boogie Procedures for ATL matched rules
VeriATL
Library for Semantics of EMF Metamodel, and Library for Semantics of OCL
Boogie Verifier
Z3 Theorem Prover
Annotations:
Code Generation
Generated components
Concrete components

Fig. 4.4 Overview of the VeriATL verification system

VeriATL aims to verify the partial correctness of ATL transformations. As its inputs it accepts the source and target EMF metamodels, a set of specified OCL contracts and an ATL transformation. The output is the result of the verification of the correctness of the ATL transformation.

Specifically, VeriATL generates the corresponding Boogie code from its inputs using three implemented code generators:

- The EMF metamodels generates Boogie types and constants using the *EMF2Boogie* code generator.
- The OCL transformation contracts produce Boogie contracts using the *OCL2Boogie* code generator.
- The ATL transformation generates Boogie procedures using the *ATL2Boogie* code generator.

The implementation details of VeriATL are described in Section 4.3.

The generated Boogie code is driven by the three core components of VeriATL, i.e. the semantics of EMF metamodels, OCL and ATL. The first two are modularised into two separate Boogie libraries (Chapter 3), which can be reused across different verifier designs. These also produce the foundation for the semantics of ATL.

Then, VeriATL sends the generated Boogie code to the Boogie verifier, and relies on Z3 to perform automatic theorem proving. Finally, if the Boogie verifier confirms the correctness of the ATL transformation with respect to the specified OCL contracts, then VeriATL simply reports that the verification is successful. Otherwise, the trace information from the Boogie verifier, indicating where the transformation incorrectness was detected, will be output.

4.2 A Semantics for ATL in Boogie

Having given a brief overview of VeriATL, we introduce one of its core component in this section, i.e. the semantics of ATL. It is based on the libraries for the semantics of EMF metamodels and OCL that we introduced in Chapter 3. In what follows, we decompose the semantics of ATL into the semantics of ATL matched rules and the semantics of ATL rule scheduling, and explain each individually.

4.2.1 Semantics of ATL Matched Rules

Each ATL matched rule specifies the mapping from the source metamodel to the target metamodel, with OCL added using both OCL data types and OCL declarative expressions. According to the language specification of ATL [6], the mapping defined by the ATL matched rule is performed in two steps: an instantiation step and an initialisation step. The semantics of each step is encoded by the Boogie procedure with a contract. The correctness of the encoding will be verified in Chapter 5.

Basics. Before introducing the Boogie encoding for the semantics of the instantiation step and the initialisation steps, there are three points that need to be emphasised:

- We introduce two functions to help the encoding. The *getTarget* function returns the corresponding target element generated for a sequence of source elements. Its inverse function *getTarget_inverse* returns the sequence of source elements used to generate the given target element.

- Our Boogie encoding addresses the **frame problem**. That is, a Boogie procedure with its contract must not only specify how it affects the transformation state, but must also manifest what memory locations it will definitely not modify. The Burstall-Bornat memory model (Section 3.2.3) helps us to deal with the frame problem. First, it allows us to quantify over all the attributes/associations, and specify the ones that are not affected by a binding operation. Second, we use different *heaps* to represent the source and target models, and axiomatise them to be disjoint (an element that is allocated on one heap is not allocated on the other heap). This ensures, for example, a modification made on the target heap will not affect the state of the source heap.

- Unlike most OO program verifiers [8, 80, 82, 114], all framing in VeriATL is done at the field granularity, not the object granularity. Compared to object granularity, the advantage of field granularity is a more detailed frame condition that specifies how the fields of model elements are changed, at the cost of more verbose contracts.

Step 1: Instantiation. The encoded Boogie contract for the instantiation step has the following structure:

- It requires that the source element(s) (as a sequence) matched by a rule are not matched by any other rule.
- It specifies that the instantiation step will only affect the heap for the target model. How the heap for the target model is affected is further specified by the following 3 postconditions.
- It ensures that after the execution of the instantiation step, for each source element(s) matched by a rule the corresponding target element(s) are allocated (but the bindings are not performed yet).
- It addresses the frame problem by ensuring that nothing else is modified, except the target element(s) created by the instantiation step.
- It further addresses the frame problem by also ensuring that each model element that was allocated before executing the instantiation step is still allocated.

Example 4.2.1. The automatically generated Boogie encoding for the instantiation step for the *S2S* rule is shown in Fig. 4.5:

- First, the precondition that specifies that the target element generated for the *ERSchema* source element is not yet allocated is expressed in the *requires* clause (line 2 - 4).
- Then, it specifies that the instantiation step will only affect the heap for the target model using the *modifies* clause (line 5).

- Next, it ensures that after the execution of the instantiation step, for each *ERSchema* element, the corresponding *RELSchema* target element is allocated (line 6 - 10).
- It also ensures that nothing else is modified, except the *RELSchema* element created from the *ERSchema* element by the instantiation step (line 11 - 15).
- Finally, it ensures that each reference that was allocated before executing the instantiation step is still allocated (line 16 - 17).

```
1   procedure S2S_matchAll();
2   // Not matched before
3   requires (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$ERSchema)⟹
4     getTarget([s])=null ∨ ¬read(tarHeap, getTarget([s]), alloc));
5   modifies tarHeap;
6   // Instantiation outcome
7   ensures (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$ERSchema)⟹
8     read(tarHeap, getTarget([s]), alloc)
9     ∧ getTarget([s]) ≠ null
10    ∧ dtype(getTarget([s]))=REL$RELSchema);
11  // Frame condition
12  ensures (∀<α> o: ref, f: Field α •
13    (o=null ∨ read(tarHeap,o,f)=read(old(tarHeap),o,f)
14    ∨ (dtype(o)=REL$RELSchema
15        ∧ f=alloc ∧ dtype(getTarget_inverse(o)[0])=ER$ERSchema)));
16  // Frame condition
17  ensures (∀ o: Ref • read(old(tarHeap),o,alloc)⟹read(tarHeap,o,alloc));
```

Fig. 4.5 The auto-generated Boogie contract for the instantiation step of the *S2S* rule

Step 2: Initialisation. The encoded Boogie contract for the initialisation step has the following structure:

- It requires that the target element(s) corresponding to source elements for a given ATL matched rule is/are instantiated.
- It specifies that only the target model heap will be modified.
- It ensures that the structural features of each target element are fully initialised, by performing associated bindings as specified in the ATL matched rule. A crucial *resolve* algorithm is performed during the bindings, which is described later. In particular:
  - If the structural feature that to be initialised is an association and its multiplicity has an upper-bound that is greater than one, then the pre-state of the structural feature composed with the result of resolve algorithm is used to initialise the structural feature.

  - Otherwise, the result of resolve algorithm is directly used to initialise the structural feature.

- It addresses the frame problem by ensuring that nothing else is modified, except the binding performed on the structural features of each target element.
- It further addresses the frame problem by also ensuring that each model element that was allocated before executing the instantiation step is still allocated.

Example 4.2.2. The Boogie contract automatically generated for the initialisation step of the *S2S* rule is shown in Fig. 4.6:

- First, it requires that the instantiation step of the *S2S* rule is finished (line 2 - 5). That is for each *ERSchema* element, the corresponding *RELSchema* target element is allocated.
- Then, it specifies that only the heap for the target model will be modified (line 6).
- Next, it ensures that the corresponding target element is fully initialised, by performing associated bindings as specified by the *S2S* rule (line 7 - 26). For example, after the initialisation step, the length of the *relations* for the *RELSchema* element equals the sum of the length of the *entities* and *relships* for the *ERSchema* (line 10 - 15). Moreover, the value of each element in the *relations* for the *RELSchema* element is the resolved result of each element in the *entities* for the *ERSchema* element (line 16 - 20), followed by the resolved result of each element in the *relships* for the *ERSchema* element (line 21 - 26).
- It also ensures that nothing else is modified, except the value of the *name* or the *relations* for the *RELSchema* element that created from the *ERSchema* element (line 27 - 32).
- Finally, it ensures that each reference that was allocated before executing the instantiation step is still allocated (line 33 - 34).

The Resolve Algorithm. The *resolve* algorithm performed during the attribute/association binding is crucial for the contract encoding. It is defined as a Boogie function, and axiomatised as follows (assume the binding is of the form lhs<-$resolve(y)$):

- If y is of a primitive type, then y is returned.
- If y is of any reference type, then one of the following is returned:

```
procedure S2S_applyAll();
// Instantiated
requires (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$ERSchema)⟹
  read(tarHeap, getTarsBySrcs([s]), alloc)
  ∧ getTarget([s]) ≠ null ∧ dtype(getTarget([s]))=REL$RELSchema);
modifies tarHeap;
... // t.name = resolve(s.name)
... // t.relations ≠ null ∧ t.relations.alloc
... // dtype(t.relations)=class._System.array
// length(t.relations)=length(s.entities)+length(s.relships)
ensures (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$ERSchema)⟹
  ArrayLength(read(tarHeap, getTarsBySrcs([s]),REL$RELSchema.relations))
  = ArrayLength(read(srcHeap,s,ER$ERSchema.entities))
    +ArrayLength(read(srcHeap,s,ER$ERSchema.relships))
);
// t.relations[j] = resolve(s.entities[j])
ensures (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$Entity)⟹
  (∀ j: int • 0≤j<ArrayLength(read(srcHeap,s,ER$ERSchema.entities))⟹
    read(tarHeap, getTarsBySrcs([s]),REL$RELSchema.relations)[j]
    =getTarsBySrcs({read(srcHeap,s,ER$ERSchema.entities)[j]})));
// t.relations[j+len(s.entities)] = resolve(s.relships[j])
ensures (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$Entity)⟹
  (∀ j: int • 0≤j<ArrayLength(read(srcHeap,s,ER$ERSchema.relships))⟹
    read(tarHeap, getTarsBySrcs([s]),REL$RELSchema.relations)
       [j+ArrayLength(read(srcHeap,s,ER$ERSchema.entities)]
    =getTarsBySrcs({read(srcHeap,s,ER$ERSchema.relships)[j]})));
// Frame condition
ensures (∀<α> o: ref, f: Field α •
  o≠null ∧ read(old(tarHeap),o,alloc)⟹
  (dtype(o)=REL$RELSchema ∧ (f=RELSchema.relations∨f=RELSchema.name)
    ∧ dtype(getTarget_inverse(o)[0])=ER$ERSchema)
  ∨ (read(tarHeap,o,f)=read(old(tarHeap),o,f)));
// Frame condition
ensures (∀ o: Ref • read(old(tarHeap),o,alloc)⟹read(tarHeap,o,alloc));
```

Fig. 4.6 The auto-generated Boogie contract for the initialisation step of the *S2S* rule

  - *y* is returned, provided its reference type is from the source metamodel, and *y* is not matched by any declared ATL matched rule
  - *y* is returned, provided its reference type is from the target metamodel
  - The corresponding target element is returned, provided its reference type is from the source metamodel, and *y* is matched by a declared ATL matched rule[1].

- If *y* is of a collection type, then all of the elements in *y* are resolved individually, and the resolved results are put together into a pre-allocated collection *col*, and *col* is returned.

[1]When more than one target element is generated by *y*, then the first target element generated by *y* is returned. For example, assuming *y* is processed by a rule with the format: **rule** $r\{$**from** $y : Y$ **to** $n : N,\ m : M\}$, then the *n* generated for *y* will be returned.

4.2.2 Semantics of ATL Rule Scheduling

According to the language specification for ATL, the ATL rules are scheduled to execute their instantiation steps before their initialisation steps, which ensures the confluence of the transformation [6]. In addition, in order to prove the correctness of the ATL transformation, the OCL transformation contracts, along with the ATL transformation are encoded into the Boogie language.

Example 4.2.3. Fig. 4.7 shows encoded rule scheduling on the *ER2REL* transformation. The transformation is verified against the OCL contract specified in Fig. 4.3. Some explanation follows:

- First, the OCL contracts are encoded as a Boogie contract. In particular, the OCL constraints on the source metamodels are encoded as Boogie preconditions (line 2 - 5), and the OCL constraints on the target metamodels are encoded as Boogie postconditions (line 7 - 10).
- Second, the rule scheduling of the ATL transformation is encoded as a Boogie implementation (line 12 - 21). The body of this Boogie implementation is a series of procedure calls to the encoded Boogie contracts for the instantiation step and the initialisation step of each ATL matched rule. The order of the calls are scheduled to execute the instantiation steps before their initialisation steps.
- Finally, the Boogie contract that represents the specified OCL contracts is paired with the Boogie implementation that represents the execution semantics of the ATL transformation. Such a pair forms a verification task, which is input to the Boogie verifier. The Boogie verifier either gives a confirmation that indicates the ATL transformation satisfies the specified OCL contracts, or trace information that indicates where the OCL contract violation is detected.

4.3 Our Implementation of VeriATL

To effectively evaluate VeriATL, we have implemented it using the model-to-text technology from MDE. The general idea of model-to-text technology is to serialise the inputs as models, then using a template-based code generation tool to produce the corresponding text. Because the inputs of VeriATL are either models or easy to extract into models, model-to-text technology is especially suitable for implementing VeriATL.

```
procedure main();
/* precondition: names are unique in the ER schema */
requires (∀ s1,s2: ref • s1∈OCLType#allInstance(srcHeap,ER$ERSchema)
 ∧ s2∈OCLType#allInstance(srcHeap,ER$ERSchema) ⟹
   read(srcHeap,s1,ER$ERSchema.name) ≠ read(srcHeap,s2,ER$ERSchema.name));
modifies tarHeap;
/* postcondition: names are unique in the REL schema */
ensures (∀ t1,t2: ref • t1∈OCLType#allInstance(tarHeap,REL$RELSchema)
 ∧ t2∈OCLType#allInstance(tarHeap,REL$RELSchema) ⟹
   read(tarHeap,t1,REL$RELSchema.name) ≠ read(tarHeap,t2,REL$RELSchema.name));

implementation main() {
/* Initialise Target model */
  call init_tar_model();
/* instantiation phase */
  call S2S_matchAll(); call E2R_matchAll(); call R2R_matchAll();
  call EA2A_matchAll(); call RA2A_matchAll(); call RA2AK_matchAll();
/* initialisation phase */
  call S2S_applyAll(); call E2R_applyAll(); call R2R_applyAll();
  call EA2A_applyAll(); call RA2A_applyAll(); call RA2AK_applyAll();
}
```

Fig. 4.7 Verifying the correctness of the *ER2REL* transformation

The template-based code generation tool we used to implement VeriATL is called *Xpand* [68]. It is chosen because of:

- Readability. *Xpand* uses explicit delimiters to enhance the readability of its template.
- Interoperability. *Xpand* is able to interact with Java for code generation tasks that are difficult to express in *Xpand*, e.g. declaring and referring to a global variable.
- Modularity. The *Xpand* templates can be organised separately and imported on demand.
- Support for evaluating OCL expressions.

Other template-based code generation tools with comparable functionality can be used to implement VeriATL.

There are three code generators in VeriATL:

- The first code generator *EMF2Boogie* reads in the input EMF metamodel of VeriATL (which is essentially a model that conforms to the *ECore* metamodel [109]). Then, it generates the corresponding Boogie types and constants from the model using our *EMF2Boogie* template. The template is about 70 lines of code written in *Xpand*.
- The second code generator *OCL2Boogie* reads in the input OCL transformation contracts of VeriATL. Then, it serialises the input into an OCL model. This is accomplished by a Java program (10 lines) that we wrote in order to interact with the OCL

extractor API (provided by the ATL compiler). Next, the corresponding Boogie contracts are produced from the OCL model using our *OCL2Boogie* template, which is about 400 lines of code written in *Xpand*.

- The third code generator *ATL2Boogie* reads in the input ATL transformation of VeriATL. Then, the ATL extractor API (provided by the ATL compiler) is used to serialise the input ATL transformation as an ATL model. Next, the ATL model generates Boogie procedures using our *ATL2Boogie* template. In particular, the *ATL2Boogie* template is sub-divided into a template (about 450 lines) that generates Boogie procedures for the instantiation step of each ATL rule, and another template (about 410 lines) that generates Boogie procedures for the initialisation step of each ATL rule.

The generated Boogie code is driven by the three core components of VeriATL, i.e. the semantics of EMF metamodels, OCL and ATL. Then, VeriATL sends the generated Boogie code to the Boogie verifier to confirm the correctness of the ATL transformation with respect to the specified OCL contracts.

4.4 Evaluation of VeriATL

In this section, we evaluate the performance of VeriATL on two case studies, i.e. the *ER2REL* and the *HSM2FSM* ATL transformations.

The *ER2REL* transformation, as shown in Fig. 4.2, translates an ER diagram to a relational schema. It is a modified version of the one originally developed by Büttner et al. [22]. The modification does not cause the ATL transformation to behave differently. However, it contains a feature (i.e. consecutive bindings in an ATL matched rule) that is not considered in the previous work. The *HSM2FSM* transformation translates a hierarchical state machine to a flattened state machine. It was originally presented by Baudry et al. to demonstrate the challenges in model transformation testing [10].

Table 4.1 summarises the two case studies in terms of *7* metrics which we use to quantify the verification complexity[2]. Most of the metrics are straightforward, simply measuring the quantity of certain constructs. The complexity of OCL is measured by counting the number of leaf nodes in the abstract syntax tree of each OCL contract. For example, the OCL expression $s\text{-}{>}select(attr : ERAttribute | attr.isKey)$ has a complexity measure of 3. The last row in Table 4.1 shows the total and average number of leaf nodes in the specified OCL contracts for each case study. The full list of the relevant metamodels, specified OCL

[2]The *mm* in the table stands for the metamodel.

transformation contracts, and the ATL model transformation for each case study can be found in Appendix B. Moreover, we refer to our online repository for the generated Boogie programs of the two case studies [30].

Metric	**ER2REL**	**HSM2FSM**
No. of Classifiers (source mm/target mm)	5/3	6/6
No. of Attributes (source mm/target mm)	6/4	2/2
No. of Associations (source mm/target mm)	6/2	5/5
No. of ATL rules	6	7
No. of ATL rule filters	3	5
No. of OCL contracts (total/pre/post)	11/7/4	27/14/13
Complexity of OCL contract (total/average)	88/8	245/9

Table 4.1 The verification complexity metrics of ER2REL and HSM2FSM case studies

Our evaluation uses the Boogie verifier (version 2.2) and Z3 (version 4.3) at the back-end of our VeriATL verification system. It is performed on an Intel 2.93 GHz machine with 4 GB of memory running on the Windows operating system. Verification times are recorded in seconds.

First, the modified *ER2REL* transformation is verified against 4 OCL postconditions that are specified by Büttner et al. VeriATL produces the same verification result as reported by Büttner et al. with the same set of OCL contracts (pre/postconditions). Specifically, 3 postconditions of *ER2REL* transformation are verified (i.e. *unique_rel_schema_-names*, *unique_rel_relation_names*, *exist_rel_relation_iskey*). The postcondition *unique_-rel_attribute_names* is not verified (which we have analysed in more detail in Appendix B.1).

Table 4.2 shows the performance of the verification of the correctness for the *ER2REL* case study. The *second* column shows the type of the OCL postconditions (e.g. syntactic or semantic correctness). The *third* column shows the size of the Boogie code automatically generated for verifying the OCL contracts (including Boogie encodings for the metamodels, the OCL contracts and the semantics of the ER2REL transformation). Its corresponding

OCL postcondition	**Type**	**Boogie (LoC)**	**Veri. Time (s)**	**Automation**
unique_rel_schema_names	semantic	885	0.359	Auto
unique_rel_relation_names	semantic	894	3.572	Semi
unique_rel_attribute_names	semantic	894	0.407	N/A
exist_rel_relation_iskey	semantic	892	0.343	Semi
Total		3565	4.681	

Table 4.2 Performance measures for verifying the transformation correctness of ER2REL

verification time is shown in the *fourth* column. In addition, we report that 2 out of 4 OCL postconditions are verified semi-automatically. This is because of incompleteness issues with our approach, which we analyse in Section 4.5.

Second, the *HSM2FSM* transformation is verified against 13 OCL postconditions that are specified by Büttner et al. [22]. VeriATL produces a different verification result as reported by Büttner et al. with the same set of OCL contracts (pre/postconditions). That is 12 postconditions are automatically verified except the postcondition *fsm_transition_src_-multi_lower*. The unverified postcondition points out a boundary case that is not considered in the work of Büttner et al. which we have analysed in more detail in Appendix B.2.

Table 4.3, structured in the same way as Table 4.2, shows the performance of the verification of the correctness for the *HSM2FSM* case study.

Compared to the verification of the *ER2REL* transformation, the main challenge of verifying the *HSM2FSM* transformation stems from the fact that the target metamodel has abstract classifiers, and the specified OCL postconditions are sometimes quantified over this abstract classifier. The tricky part is to establish the postcondition on each sub-classifier in order to conclude the postcondition holds on the abstract classifier. Interactive theorem provers, such as Coq, usually perform a manual structural induction (i.e. a case analysis on all the sub-classifiers of the abstract classifier) for this task, which is labour-intensive and time consuming (because most of the cases are treated similarly). Our evaluation on the *HSM2FSM* transformation shows that VeriATL's underlying automatic theorem prover, i.e. the Z3 SMT solver, is able to figure this out automatically.

OCL postcondition	**Type**	**Boogie (LoC)**	**Veri. Time (s)**	**Automation**
unique_fsm_sm_names	semantic	1092	6.599	Auto
unique_fsm_state_names	semantic	1092	3.386	Auto
fsm_state_multi_lower	syntactic	1092	2.792	Auto
fsm_state_multi_upper	syntactic	1100	2.547	Auto
fsm_transition_multi_lower	syntactic	1092	3.037	Auto
fsm_transition_multi_upper	syntactic	1100	2.389	Auto
fsm_transition_src_multi_lower	syntactic	1092	2.613	N/A
fsm_transition_src_multi_upper	syntactic	1100	2.609	Auto
fsm_transition_trg_multi_lower	syntactic	1092	2.723	Auto
fsm_transition_trg_multi_upper	syntactic	1100	2.527	Auto
fsm_transition_src_contain_sm	syntactic	1108	3.660	Auto
fsm_transition_trg_contain_sm	syntactic	1108	3.741	Auto
fsm_transition_contain_sm	syntactic	1108	4.832	Auto
Total		14276	43.455	

Table 4.3 Performance measures for verifying the transformation correctness of HSM2FSM

4.5 Limitations of VeriATL

The evaluation strongly demonstrates the feasibility of VeriATL. However, VeriATL currently has some limitations.

Soundness. The soundness of VeriATL depends on the consistency of our Boogie libraries for the semantics of EMF metamodels and OCL (as discussed in Section 3.4). At the moment, our Boogie libraries for the semantics of EMF metamodels and OCL are structural, intuitive and available for inspection. In addition, we have designed a regression test suite with test oracles that specifies verification scenarios and their expected outcome. The regression test suite is executed on every modification to the Boogie libraries, or modifications to the Boogie code compilation process (e.g. OCL compilation, ATL rules compilation to Boogie). This is to ensure the soundness of VeriATL on each rebuild.

Completeness. The completeness of VeriATL remains one of the major concerns. The incompleteness could be due to our encodings (as discussed in Section 3.4). It could also be due to known limitations of SMT solvers when working with quantifiers [46, 53, 81, 84]. Our experience with VeriATL shows that Z3 is not able to efficiently handle formulas involving existential quantifiers (e.g. *exist_rel_relation_iskey* contract in *ER2REL* transformation). When the existential quantifier cannot be automatically proved by Z3, our experience shows that three techniques can help:

1. Rewrite the existential quantifier to its equivalent universal quantifier.
2. Encapsulate the quantifier body into a Boogie function.
3. Provide a witness for the existential quantifier.

The first technique is a general approach for all data types. The later two are especially useful when the quantifications are over the integer domain.

Example 4.5.1. In Boogie, developers can use the *assert* statement to prescribe a proof at a specific execution point of a Boogie program. If its operand evaluates to true on the execution point to prove, the *assert* statement simply reduces to no operation. Otherwise, the Boogie program ends up in an irrecoverable *error* state on proof failure. The following simple Boogie assertion statement cannot be directly proved by Z3.

```
procedure test(x: int)
{ assert (¬(∃ n: int • x div 2 = n )) ⟺ (∀ n: int • x div 2 ≠ n); }
```

However, using the second technique, which encapsulates the quantifier body $x\ div\ 2 == n$ into a Boogie function *f*, produces the following Boogie code, which can be automatically proved:

```
function f (x: int, n: int) : bool
{ n = x div 2 }

procedure test (x: int)
{ assert (¬(∃ n: int • ¬f(x,n))) ⟺ (∀ n: int • f(x, n)) ; }
```

Example 4.5.2. Consider the following Boogie program which is not verified because of the incompleteness of underlying theorem prover (even though the second technique is used):

```
function f (x: int, n: int) : bool
{ n = x div 2 }

procedure test (x: int)
{ assert (∃ n: int • f(x, n)); }
```

Using the third technique, by manually providing a witness $f(x,x\ div\ 2)$, produces the following Boogie code, which can be automatically proved:

```
function f (x: int, n: int) : bool
{ n = x div 2 }

procedure test (x: int)
{ assert f(x, x div 2);
  assert (∃ n: int • f(x, n)); }
```

ATL coverage. VeriATL covers the declarative aspect of ATL, i.e. ATL matched rules. It supports one-to-one mappings of (possibly abstract) classifiers with the default *resolve* algorithm. However, VeriATL can be extended with advanced ATL features, such as lazy rules (lazy rules are called from the other rules, which are not as frequently used as the matched rules but are the main source of transformation non-termination) and imperative features. For example, we have developed a small toy ATL transformation for refactoring Java fields with certain annotations [30]. It demonstrates how to use VeriATL to verify one-to-many ATL transformations with imperative constructs such as user-controlled resolution *resolveTemp*. In the future, we would like to cover more ATL features to build upon the current VeriATL verifier.

Expressiveness. Because of the underlying SMT solver, the expressiveness of transformation contracts is based on FOL with equality. To ensure this expressiveness power is useful in practice for MTr verification, we need to experiment with more ATL transformations that have OCL contracts specified.

4.6 Summary

In this chapter we have introduced VeriATL, our modular and reusable verifier for verifying the correctness of the ATL transformations. We have presented the semantics of ATL, based on the two modular Boogie libraries for the semantics of EMF and OCL presented in Chapter 3. We have evaluated VeriATL on two case studies, showing its performance and feasibility. The limitations of VeriATL and lessons learned from its development are also discussed.

Notice that the semantics of ATL that we encoded in this chapter is essentially an execution semantics of ATL that is abstracted at a coarse granularity. It describes what the ATL transformations are trying to achieve. However, it supports neither its soundness proofs, nor termination verification, both of which are important aspects of advanced verifier design. Therefore, the missing piece of VeriATL is a fine grained semantics of ATL that explains the runtime behaviour of ATL (i.e. how ATL actually works at runtime). In the next chapter, we will investigate the semantics of ATL at a lower level, i.e. the compiled ASM byte code, to realise the missing piece.

Chapter 5

A Sound Execution Semantics for ATL via a Translation Validating ASM Implementation

In this chapter, we present a translation validation approach to verify that the execution semantics of ATL, encoded by VeriATL, soundly represents the runtime behaviour of its corresponding compiled implementation in terms of bytecode instructions for the ASM. In Section 5.1, we introduce the basic definitions for the sound execution semantics of ATL, and describe how to enhance VeriATL with a translation validation approach for this task. We detail the procedure of the translation validation approach in Section 5.3. The core component of this approach is the semantics of the ASM bytecode (Section 5.2), which is the main contribution of this chapter. In Section 5.4, we detail our implementation for VeriATL with a translation validation approach. Then, in Section 5.5, we evaluate our approach on two ATL transformations, to show its performance and feasibility. This chapter concludes with a discussion of the lessons learned from applying our translation validation approach (Section 5.6).

5.1 Introduction

Ab.Rahim and Whittle, in their survey, identify that ensuring the semantics preservation relationship between a declarative specification and its operational implementation is an under-researched area in MDE [1]. They find that existing model transformation verification approaches do not verify that the encoded execution semantics of a transformation specification soundly represents the runtime behaviour of a transformation implementation. As a result, an unsound encoding will yield unsound results after verification, i.e. it will lead to erroneous conclusions about the correctness of the ATL transformation.

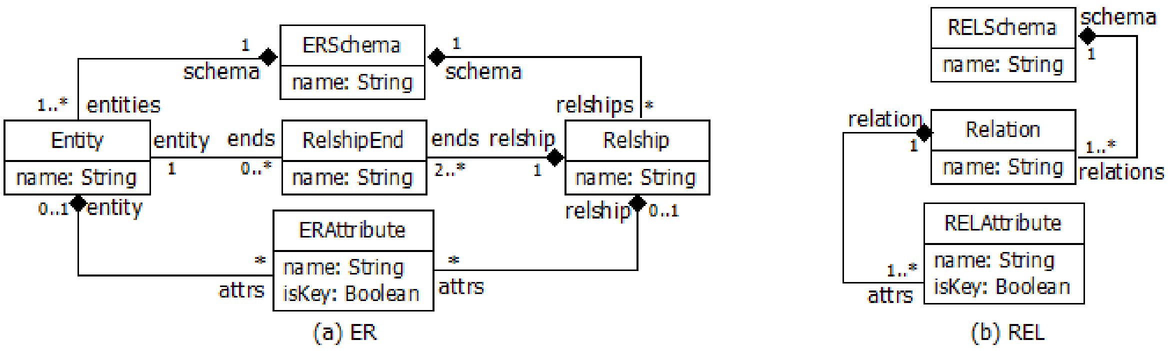


Fig. 5.1 Entity-Relationship and Relational metamodels

VeriATL, presented in Chapter 4, could suffer from the same symptom. We demonstrate this using the *ER2REL* transformation that was given in Chapter 4. Both the *ER* and *REL* metamodels (Fig. 5.1), and the *ER2REL* transformation remain unchanged (Fig. 5.2).

```
module ER2REL; create OUT : REL from IN : ER;

rule S2S {
  from s: ER!ERSchema
  to t: REL!RELSchema (name <- s.name, relations <- s.entities, relations <- s.relships} )}

rule E2R {
  from s: ER!Entity to t: REL!Relation ( name <- s.name) }

rule R2R {
  from s: ER!Relship to t: REL!Relation ( name <- s.name) }

rule EA2A {
  from att: ER!ERAttribute, ent: ER!Entity (att.entity = ent)
  to t: REL!RELAttribute ( name <- att.name, isKey <- att.isKey, relation <- ent ) }

rule RA2A {
  from att: ER!ERAttribute, rs: ER!Relship ( att.relship = rs )
  to t: REL!RELAttribute ( name <- att.name, isKey <- att.isKey, relation <- rs ) }

rule RA2AK {
  from att: ER!ERAttribute, rse: ER!RelshipEnd
      ( att.entity = rse.entity and att.isKey = true )
  to t: REL!RELAttribute ( name <- att.name, isKey <- att.isKey, relation <- rse.relship )}
```

Fig. 5.2 ATL transformation for *ER2REL* model transformation

Using the OCL contract specified in Fig. 5.3, the goal is to verify whether the *unique_-er_relship_names* constraint (i.e. all instances of *ERSchema* have unique names for its *relships*) imposed on the *ER* metamodel, along with the *ER2REL* transformation, guarantees that the *unique_rel_relation_names* constraint (i.e. all instances of *RELSchema* have unique names for its *relations*) holds on the *REL* metamodel.

```
context ER!ERSchema inv unique_er_relship_names: -- relship names are unique in the ER schema
  ER!ERSchema.allInstances()->forAll(s | s.relships->forAll(r1,r2 | r1<>r2 implies r1.name<>r2.name))
-----------------------------------------
context REL!RELSchema inv unique_rel_relation_names: -- relation names are unique in RELSchema
  REL!RELSchema.allInstances()->forAll(s | s.relations->forAll(r1,r2| r1<>r2 implies r1.name<>r2.name))
```

Fig. 5.3 OCL contracts for ER and REL

Whether the ER2REL transformation is verified for the given OCL contracts depends on how the Boogie contracts for the execution semantics of each ATL matched rule are encoded. Our Boogie encoding of VeriATL is based on the existing documentation of ATL [6, 66]. However, the ambiguities in the documentation increase the encoding difficulty. For example, on line *5* of the ER2REL transformation (Fig. 5.2), the *relations* association is bound twice. The ATL documentation does not explicitly specify how to encode the execution semantics of such a case. It can be encoded by either assuming that:

- The second binding *overwrites* the first binding. In this case the *unique_rel_relation_-names* constraint holds, since the *relations* of each *RELSchema* element will be resolved from the *relships* of the *ERSchema* element only; or
- The second binding is *composed* with the first binding. In this case the *unique_-rel_relation_names* constraint does not hold, since the *relations* of each *RELSchema* element will come from both the *entities* and *relships* of the *ERSchema* element. We do not know that the names of *relships* are all unique for each *ERSchema* element, nor that the names of *entities* and *relships* of each *ERSchema* element are different.

Adapted VeriATL Verification System. To resolve the ambiguity here, our quest in this chapter is to adapt VeriATL to use a translation validation approach. The goal is to demonstrate how to compositionally verify the termination and the soundness of the Boogie encoding of the execution semantics of each ATL matched rule in the given ATL transformation. In this work, verifying soundness means verifying that the execution semantics of each ATL matched rule soundly represents the runtime behaviour of its corresponding ASM implementation.

Consequently, VeriATL can soundly verify the correctness of an ATL transformation against its specified OCL contracts, based on the sound encodings for the execution semantics of the ATL matched rules.

The adapted architecture of VeriATL is shown in Fig. 5.4. An additional translation validation layer is added in VeriATL to verify the soundness of the Boogie encoding for the

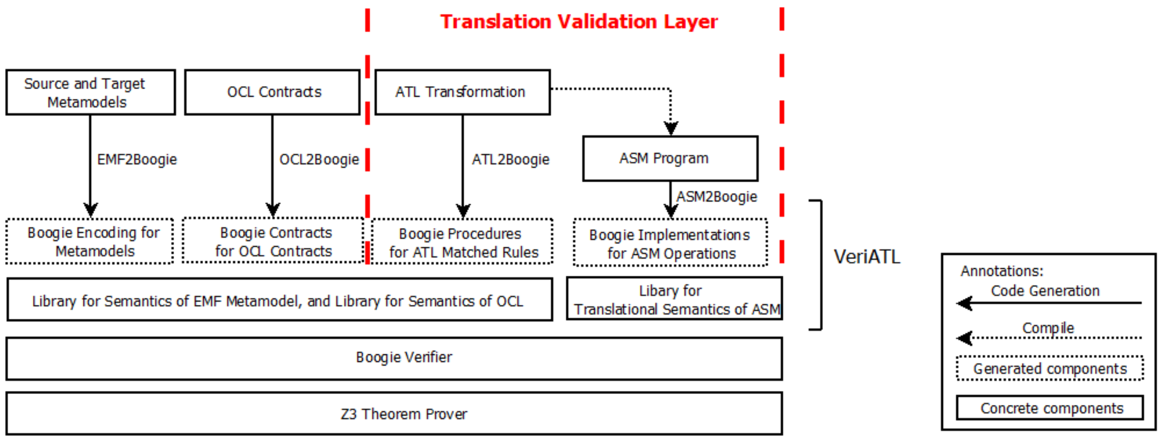

Translation Validation Layer
Source and Target Metamodels
OCL Contracts
ATL Transformation
ASM Program
EMF2Boogie
OCL2Boogie
ATL2Boogie
ASM2Boogie
Boogie Encoding for Metamodels
Boogie Contracts for OCL Contracts
Boogie Procedures for ATL Matched Rules
Boogie Implementations for ASM Operations
Library for Semantics of EMF Metamodel, and Library for Semantics of OCL
Libary for Translational Semantics of ASM
Boogie Verifier
Z3 Theorem Prover
VeriATL
Annotations:
Code Generation
Compile
Generated components
Concrete components

Fig. 5.4 Overview of the adapted VeriATL verification system

execution semantics of the ATL transformation. The additional layer accepts two inputs, i.e. the source and target EMF metamodels, and an ATL transformation. The output is an execution semantics of the ATL transformation encoded in Boogie, which correctly represents the runtime behaviour of its corresponding ASM implementation. As a result, the verification for the correctness of an ATL transformation that is based on the output of the translation validation layer will be sound.

Specifically, the translation validation layer generates the corresponding Boogie code from its inputs using three implemented code generators:

- The EMF metamodels generate Boogie types and constants using our *EMF2Boogie* code generator (presented in Section 4.3).
- The ATL transformation generates Boogie procedures using our *ATL2Boogie* code generator (presented in Section 4.3).
- The ATL compiler is used to compile the input ATL transformation into an ASM program. This compiled ASM program is then used to generate Boogie implementations using our *ASM2Boogie* code generator.

The implementation detail of the *ASM2Boogie* code generator is illustrated in Section 5.4.

The generated Boogie code is driven by three core components of adapted VeriATL. That is the semantics of the EMF metamodel, ATL and ASM. The first two have already been presented in Chapter 3 and Chapter 4. The third one encapsulates the translational semantics of the ASM language in a Boogie library, to precisely explain the runtime behaviour of ASM implementations.

VeriATL sends the generated Boogie code to the Boogie verifier to automatically prove the soundness of our Boogie encoding. Finally, if the Boogie verifier confirms that the execution semantics of an ATL transformation encoded in Boogie is sound, then this encoding will be output by VeriATL. Otherwise, the trace information from the Boogie verifier, indicating where the encoding unsoundness was detected, will be output.

In the next sections, we describe our translation validation approach to verify the soundness of the Boogie encoding for the execution semantics of ATL transformations.

5.2 A Translational Semantics for ASM

Our translation validation approach is based on providing a translational semantics of the ASM language in Boogie, which allows the runtime behaviour of the ASM implementation to be represented using Boogie implementations.

We define this translational semantics of the ASM language via a list of translation rules to Boogie. Each translation rule encodes the operational semantics of an ASM instruction in Boogie.

Specifically, the ASM language contains 21 bytecode instructions. Apart from the general-purpose instructions for stack handling and control flow, an important feature of the ASM language is the model-handling-specific instructions that are dedicated to model manipulation.

The only resource we can find to explain the operational semantics of ASM bytecode instructions is the specification of the ATL virtual machine [6]. However, it is imprecise and leaves many issues open (Section 5.6.1). This raises the question of how a correct translation rule, especially for each model handling instruction, should be encoded in Boogie.

Unlike the other two categories of instructions, the model handling instructions can have different operational semantics for different model management systems. This is because ATL aims at interacting with a range of model management systems which offer different interfaces for model manipulation [66].

Our strategy is to focus on the EMF model management system. Then, we can check the ATL source code (specifically the ATL virtual machine implementation that relates to EMF) for the operational semantics of each ASM instruction, and then design the translation rule correspondingly.

An ASM implementation contains a list of ASM operations. Each operation has a list of local variables, which are encoded as Boogie local variables. An operand stack is used by each ASM operation to communicate values for local computations. This is abstracted as an

ASM Instruction (S)	**Corresponding Boogie Statements ([[S]])**
push$_\tau$ *c*	Stk := [[*c*]]::Stk ; (where *c* is a constant of type $\tau \in \{int, bool, string\}$)
pop	`assert` size(Stk)>0 ; Stk := tl(Stk) ;
store *x*	`assert` size(Stk)>0 ; *x* := hd(Stk) ; Stk := tl(Stk) ; (where *x* is a variable)
load *x*	Stk := *x*::Stk ; (where *x* is a variable)
swap	`assert` size(Stk)>1 ; Stk := hd(tl(Stk))::hd(Stk)::tl(tl(Stk)) ;
dup	`assert` size(Stk)>0 ; Stk := hd(Stk)::Stk ;
dup_x1	`assert` size(Stk)>1 ; Stk := hd(Stk)::hd(tl(Stk))::hd(Stk)::tl(tl(Stk)) ;

Table 5.1 Translational semantics for ASM stack handling instructions

OCL *sequence* data type, which is represented as a list in Boogie called *Stk* in our encoding. Source and target elements are globally accessible by every ASM operation, and they are managed by the disjoint source and target *heaps* as described in Chapter 4.

The full translational semantics of the ASM language is given in Table 5.1 - Table 5.3, classified by the category that each ASM instruction resides in. In what follows, we pick a representative ASM instruction from each category, and explain the intuition behind its corresponding translation rule.

Stack Handling Instructions. The translational semantics of the ASM stack handling instructions is shown in Table 5.1. The *STORE* instruction is one of stack handling instructions. It has one operand which is a local variable that the instruction operates on. The stack is expected to be non-empty for the instruction to succeed, since it assigns the top of the stack to its operand. After the assignment, the top of the stack is then dropped. Such operational semantics for the *STORE* instruction is encoded by its corresponding translation rule in Boogie as shown in Table 5.1. In our Boogie encoding, to make sure the operand of the *STORE* instruction is declared before use, we generate a Boogie variable (in the same scope of the encoded *STORE* instruction) for each local variable of an ASM operation with unique name and equivalent type.

Control Flow Instructions. The translational semantics of the ASM control flow instructions is shown in Table 5.2. The conditional instruction *IF* is one of the control instructions, which formalizes a case distinction according to the popped boolean value of the operand stack. If the popped value is true, the ASM operation continues at the instruction identified by the operand of the *IF* instruction. Otherwise, the *IF* instruction reduces to no operation. The presented translation rule encodes this operational semantics for the *IF* instruction. In our Boogie encoding, to make sure the offset of the *IF* instruction points to a valid program point, we insert a fresh Boogie label at the program point which corresponds to the offset of the *IF* instruction.

Model Handling Instructions. The translational semantics of the ASM model handling instructions is shown in Table 5.3. The *SET* instruction is one of the ASM instructions for

ASM Instruction (S)	**Corresponding Boogie Statements (⟦S⟧)**
if *n*	`var` cond$^{\#}$: bool ; `assert` size(Stk) > 0 ; cond$^{\#}$:= hd(Stk) ; Stk := tl(Stk) ; `if` (cond$^{\#}$) `goto` *l* ; (where *l* is a fresh label. It labels the program point which corresponds to the ASM instruction offset *n*)
goto *n*	`goto` *l* ; (where *l* is a fresh label. It lables the program point which corresponds to the ASM instruction offset *n*)
iter $Stmt_1$ **enditer**	`var` col$^{\#}$: Seq ref ; `assert` size(Stk) > 0 ; col$^{\#}$:= hd(Stk) ; Stk := tl(Stk) ; `while` (hasNext(col$^{\#}$)) *INV* { Stk := next(col$^{\#}$)::Stk ; ⟦$Stmt_1$⟧}
pcall *sig*	let n = arg_size(sig) in $let\ \overline{args} = tk(Stk,n),\ ctx = hd(dp(Stk,n))\ in$ `assert` size(Stk) > *n* ; `call` invoke(reflect(*sig*, *ctx*), $\overline{args}$) ; Stk := dp(Stk, *n*+1) ;
call *sig*	let n = arg_size(sig) in $let\ \overline{args} = tk(Stk,n),\ ctx = hd(dp(Stk,n))\ in$ `var` result$^{\#}$: *T* ; `assert` size(Stk) > *n* ; `call` result$^{\#}$:= invoke(reflect(*sig*, *ctx*), $\overline{args}$) ; Stk := result$^{\#}$::dp(Stk, *n*+1) ; (where *T* is the return type of the reflected method)

Table 5.2 Translational semantics for ASM control flow instructions

model handling. The parameter of a *SET* instruction is a structural feature f (either an attribute or an association). Before executing the *SET* instruction, the top two elements on the operand stack are an element o (second-top) and a value v (top) respectively.

The operational semantics of the *SET* instruction forms a case distinction according to the instruction parameter f: if f is an association and its multiplicity has an upper-bound that is greater than one, then compute the union of the value of $o.f$ with v; otherwise, set $o.f$ to v. Finally, the top two elements are popped.

Thus, the operational semantics of the *SET* instruction explains the unusual behaviour of consecutive bindings to the *relations* association (whose multiplicity has an upper-bound that is greater than one) shown on line 5 of Fig. 5.2. Each binding corresponds to a *SET* instruction on the ASM level. Therefore, the two consecutive bindings correspond to two *SET* instruction invocations. The result is a composition of two bindings.

The translation rule for the *SET* instruction offers no surprise from its operational semantics, except for two points:

- Since we use different *heaps* to represent the source and target models, the *heap* that the *SET* instruction operates on is determined by the data type of second-top element of the operand stack. This is accomplished by the *ASM2Boogie* code generator (Section 5.4).

ASM Instruction (S)	**Corresponding Boogie Statements ($[\![S]\!]$)**
new *r*	let mm = hd(Stk), cl = hd(tl(Stk)) in $let\ clazz = resolve(mm,cl)\ in$: `var` $r^{\#}$: Ref ; `havoc` $r^{\#}$; `assume` $r^{\#} \neq null \wedge \neg$read(heap, $r^{\#}$, $alloc$) ; `assert` size(Stk) > 1 ; `assume` typeof($r^{\#}$) = $clazz$; heap := update(heap,$r^{\#}$,$alloc$,$true$); Stk := $r^{\#}$::tl(tl(Stk)) ;
get *f*	let o = hd(Stk) in `assert` size(Stk) $> 0 \wedge o \neq null \wedge$ read(heap, o, $alloc$) ; Stk := read(heap,o,f)::tl(Stk) ;

set *f*	let o = hd(tl(Stk)), v = hd(Stk) in
	`assert` size(Stk) $> 1 \wedge o \neq null \wedge$ read(heap, o, $alloc$) ;
	`if` (isCollection(f)) { heap := update(heap,read(heap,o,f),read(heap,o,f) $\cup$ v); }
	`else` { heap := update(heap,o,f,v) ; }
	Stk := tl(tl(Stk));

findme	let mm = hd(Stk), cl = hd(tl(Stk)) in `assert` size(Stk) > 1 ; Stk := resolve(mm, cl)::tl(tl(Stk)) ;
getasm	Stk := ASM::Stk ;

Table 5.3 Translational semantics for ASM model handling instructions

- An *isCollection* function (of type $Field\ \alpha \rightarrow\ bool$) is encoded while mapping the structural features of classifiers to the Boogie constants. It is axiomatised so that it returns *true* when the given structural feature is an association and its multiplicity has an upper-bound that is greater than one, and returns *false* otherwise.

Finally, the full translational semantics of the ASM language is encapsulated as a Boogie library, which can be adapted by the verifier designer for other model transformation languages.

5.3 Translation Validation of Encoding Soundness

Each ATL matched rule is actually compiled into two ASM operations by the ATL compiler, i.e. a *matchAll* operation (for the instantiation step) and an *applyAll* operation (for the initialisation step). The important contribution of this chapter is the verification of the soundness of our Boogie encoding for the execution semantics of the ATL rules, i.e. that the encoded execution semantics of each ATL rule correctly represents the runtime behaviour of its corresponding ASM operation.

In order to verify the soundness of our Boogie encoding of the execution semantics of each ATL matched rule, we define the execution semantics of an ATL matched rule encoded in Boogie as being sound, if,

- the Boogie contract that represents the execution semantics of its *instantiation* step is satisfied by the Boogie implementation that represents the runtime behaviour of its *matchAll* operation, and

- the Boogie contract that represents the execution semantics of its *initialisation* step is satisfied by the Boogie implementation that represents the runtime behaviour of its *applyAll* operation.

Each of these conditions form a verification task that is processed by the Boogie verifier. If none of the verification tasks generate any errors (from the verifier), we conclude that our Boogie encoding for the execution semantics of the ATL matched rules is sound. Essentially, our approach is based on a translation validation technique used in compiler verification [97]. The benefit is that we do not need to verify that the encoded execution semantics of ATL transformations are always sound with respect to the runtime behaviour of their ASM implementation (which is difficult to automate). Instead, we can automatically verify the soundness of each ATL transformation/ASM implementation pair.

Finally, we can conclude that the execution semantics of an ATL transformation encoded in Boogie is sound when the execution semantics of all the relevant ATL matched rules encoded in Boogie are sound.

Example 5.3.1. We demonstrate our approach on the instantiation step of the *S2S* rule (Fig. 5.5).

```
procedure S2S_matchAll();        //Contract for instantiation step
...
ensures (∀ s: ref • s∈OCLType#allInstance(srcHeap,ER$ERSchema)⟹
    dtype(getTarget([s]))=REL$RELSchema);

implementation S2S_matchAll() //Implementation for matchAll operation
{ ...
  #ERSchemas := OCLType#allInstance(srcHeap,ER$ERSchema);
  counter:=0;

  while(counter <|#ERSchemas|)
    ...
    invariant (∀ n: int • 0≤n<counter⟹
      dtype(getTarget({#ERSchemas[n]}))=REL$RELSchema);
    ...
    decreases |#ERSchemas|−counter;
  { ... counter:=counter+1; }
}
```

Fig. 5.5 Verification of the soundness of Boogie encodings for the instantiation step of the *S2S* rule

Some explanation is in order. First, a Boogie implementation that contains loops is difficult to verify because the users cannot generally predict how many times the loop executes, or whether it will terminate. The key ingredient to prove the correctness of a loop is to provide the **loop invariant** (using the *invariant* clause) that is true immediately before, and immediately after each iteration of the loop. The general loop invariant for the Boogie implementation is automatically generated. This is demonstrated in the verification of the instantiation step of the *S2S* rule as follows (Fig. 5.5): In the Boogie implementation for its *matchAll* operation, an invariant is generated to ensure that for all the matched source elements that have been iterated, the postcondition of the instantiation step is fulfilled (line 13 - 14). Thus, by the end of the iteration, all the matched source elements are iterated, and therefore the postcondition of the instantiation step can be established (line 3 - 4).

Second, we also use a **variant expression** to ensure that the loop terminates. A default variant expression (automatically generated) for the Boogie implementation of a *matchAll* operation is the size of the iterated collection minus the corresponding loop counter (line 16). Since the counter increases on each iteration and the size of the processed collection remains unchanged, we can deduce that there are less elements in the collection to be iterated upon. In addition, since the loop counter has to be smaller than the length of the iterated collection to keep the loop unrolling, this ensures the variant expression is maintained above a bound (i.e. a lower bound of zero) so that the variant expression is not decreasing forever. Notice that the *decreases* clause on line 16 is not actually supported in Boogie. It is only used to demonstrate the concept of the variant expression. In practice, we record the old value of the *decreases* clause when entering the loop. Then, right after the corresponding counter of the loop increases, we check that: (a) it is greater than the current value of the *decreases* clause; and (b) it is greater or equal to the lower bound of zero.

The verification of the soundness of Boogie encodings for the initialisation step of ATL rules is performed in a similar way to the instantiation step.

5.4 Our Implementation of Adapted VeriATL

To effectively evaluate our translation validation approach, we have implemented VeriATL with an additional translation validation layer. It interacts with three code generators to generate Boogie code for performing our translation validation approach:

- The EMF metamodels generate Boogie types and constants using our *EMF2Boogie* code generator.
- The ATL transformation generates Boogie procedures using our *ATL2Boogie* code generator.
- The ASM program generates Boogie implementations using our *ASM2Boogie* code generator.

The first two code generators have been presented in Section 4.3. In this section, we briefly describe our implementation of the *ASM2Boogie* code generator.

The *ASM2Boogie* code generator is implemented in Java (about 1450 lines of code). The reason we do not use *Xpand* (as we did for the *EMF2Boogie* and *ATL2Boogie* code generators) is because currently *Xpand* (ver. 2.1) is not able to pass the generated result as an argument to the other *Xpand* templates. Consequently, tasks such as invariant generation are made unnecessarily complicated by using *Xpand* (since the loop invariants of an outer loop need to be reused by an inner loop when the loop goes deeper).

The main challenge of developing *ASM2Boogie* stems from the fact that each ASM operation is based on a generic operand stack. This would compromise the precision of our generated Boogie implementations. For example, a *GET name* instruction simply retrieves the *name* attribute for the top model element on the operand stack. However, because we use separate heaps to represent the input and output models, the type of the top model element on the operand stack is important in order for *ASM2Boogie* to generate corresponding Boogie code. Thus, we also maintain a type stack during the code generation for each ASM instruction. For example, the *PUSHI* instruction has the effect of pushing an integer onto the type stack, so that its next instruction can look up the type stack and query its state.

The generated Boogie code is driven by three core components of adapted VeriATL: the semantics of the EMF metamodel, ATL and ASM. Then, adapted VeriATL sends the generated Boogie code to the Boogie verifier to automatically prove the soundness of our Boogie encoding.

5.5 Evaluation of Adapted VeriATL

In this section, we evaluate the performance of our translation validation approach on two case studies, i.e. the *ER2REL* and the *HSM2FSM* transformations that were used in Chapter 4. We refer to our online repository for the generated Boogie programs for the two case studies [30].

Our experiment uses the Boogie verifier (version 2.2) and Z3 (version 4.3) at the backend of our adapted VeriATL verification system. It is performed on an Intel 2.93 GHz machine with 4 GB of memory running on Windows. Verification times are recorded in seconds.

Table 5.4 shows the performance for automatically verifying the soundness of our Boogie encoding for the *ER2REL* transformation. The *second* and *third* columns show the size of the Boogie code generated for the instantiation and initialisation step of the ATL matched rule respectively (shown by lines of Boogie contract/implementation). Their corresponding verification time is shown in the *fourth* and *fifth* columns.

Rule	Boogie (LoC)		Veri. Time (s)		Automation	Termination
	Inst.	Init.	Inst.	Init.		
S2S	28/74	63/78	0.125	0.187	Auto	Yes
E2R	22/74	50/40	0.125	0.047	Auto	Yes
R2R	22/74	50/40	0.125	0.047	Auto	Yes
EA2A	24/119	56/71	0.218	0.140	Auto	Yes
RA2A	24/119	56/71	0.234	0.125	Auto	Yes
RA2AK	24/133	57/75	0.296	0.110	Auto	Yes
Total	144/593	332/375	1.123	0.656		

Table 5.4 Performance measures for verifying the encoding soundness of ER2REL

Table 5.5 shows the performance for automatically verifying the soundness of our Boogie encoding for the *HSM2FSM* transformation. It is structured in the same format as Table 5.4. We report that all the verification tasks are verified automatically, and that the ATL rules for both *ER2REL* and *HSM2FSM* transformations are verified as terminating.

Rule	Boogie (LoC)		Veri. Time (s)		Automation	Termination
	Inst.	Init.	Inst.	Init.		
SM2SM	22/74	50/40	0.125	0.047	Auto	Yes
RS2RS	22/74	52/55	0.125	0.062	Auto	Yes
IS2IS	22/87	53/55	0.109	0.078	Auto	Yes
IS2RS	22/88	52/55	0.296	0.062	Auto	Yes
T2TA	22/102	56/85	0.701	0.125	Auto	Yes
T2TB	28/213	66/92	1.248	0.203	Auto	Yes
T2TC	28/213	66/92	1.342	0.203	Auto	Yes
Total	166/851	395/474	3.946	0.780		

Table 5.5 Performance measures for verifying the encoding soundness of HSM2FSM

5.6 Analysis of Our Results

The evaluation allows us to certify the soundness for our encoded execution semantics of ATL, with respect to the runtime behaviour of its ASM implementation. In this section, we discuss the lessons learned from applying our translation validation approach. The discussion is categorised into: (a) the under-specification of the ATL language and (b) why the translation validation approach is particularly suitable to ensure the encoding soundness for VeriATL.

5.6.1 Under-specification of the ATL Language

Through the evaluation, we identify a number of under-specified cases in the language specification of ATL [6]. These cases are open to interpretation and can thus pose challenges to verify the soundness of VeriATL. Therefore, we document and discuss these under-specified cases in this section.

Source Pattern Matching

The first under-specified case resides in the source pattern matching. Recall that, in ATL, the source element *e* to be matched of a certain type *t* is syntactically written as *e: t*. The ambiguity here is whether a source element should be considered matched if it is exactly of type *t* or if it is of kind *t* (i.e. *OCLIsTypeOf* or *OCLIsKindOf*: see Section 3.3 for the difference between these two operations).

Example 5.6.1. Assuming *MMA!Subtype* is a classifier inherited from *MMA!Type*, and the parent classifier is not an abstract classifier.

```
rule A { from s: MMA!Type ...}

rule B { from s: MMA!Subtype ... }
```

Fig. 5.6 Ambiguity in source pattern matching

Recall that during the *initialisation* step, a compulsory requirement is that the source pattern should not have been previously matched by other ATL rules (as described in Section 4.2.1). A violation of this requirement is called a rule conflict, and will cause a runtime exception [6]. The ATL transformation shown in Fig. 5.6 may or may not contain rule conflicts depended on how the source pattern matching is interpreted:

- Interpreted as *OCLIsTypeOf*: the elements matched by the two rules are disjoint, and thus contain no rule conflict.
- Interpreted as *OCLIsKindOf*: the elements matched by the two rules are overlapping, since the element matched by rule *A* can potentially be also matched by rule *B*. Thus, the ATL transformation contains a rule conflict.

Through our translation validation approach we have examined the ASM implementation of the *matchAll* operation, and confirm that the second interpretation is correct.

Consecutive binding

The second under-specified case arises when invoking consecutive bindings to the same association (e.g. bind *relations* association twice in Fig. 5.2). As illustrated in Section 5.2, by examining the semantics of the *SET* instruction, the consecutive bindings correspond to consecutive *SET* instruction invocations. The result will be a composition of two bindings.

Resolve algorithm

The third under-specified case is in the *resolve* algorithm. The language specification of ATL fails to specify two boundary cases:

- If *y* is of any reference type, *y* is returned, provided its reference type is from the source metamodel, and *y* is not matched by any declared ATL matched rule; or
- If *y* is of any reference type, *y* is returned, provided its reference type is from the target metamodel.

These were found in the ASM implementation of the *resolve* algorithm while we performed the translation validation approach.

5.6.2 Suitability of the Translation Validation Approach for VeriATL

The first reason for the compatibility of the translation validation approach to VeriATL is that the translation validation approach is inherently efficient. That is, the soundness encoding only needs to be verified once for each compilation to ensure the encoded execution semantics of each ATL transformation soundly represents the runtime behaviour of its corresponding ASM implementation. Such soundly encoded execution semantics of ATL transformations can be reused in order to verify the correctness of ATL transformations against their specified OCL contracts, as long as the source ATL transformation does not change.

The second reason is due to the language features of ATL:

1. Each ATL rule is written in a declarative style, and has a unified deterministic goal to achieve (i.e. mapping). Thus, it is easier to abstract the semantics of ATL rules into FOL expressions than to abstract the semantics of an imperative program that achieves an arbitrary goal. This feature greatly reduces the complexity of adapting the translation validation approach, and enables its automation.

2. We consider ATL matched rules, which are always propagated on an initially empty target model that is disjoint from the source model. Thus, we are able to use two separate *heaps* to organise the source and target elements. This ensures, for example, a modification made on the target *heap* will not affect the state of the source *heap*. Therefore, it yields a simple encoding that contributes to the automation of the translation validation approach.

3. For ATL matched rules, the iteration in the bytecode implementation always interacts with collections. Thus, we are able to automatically infer suitable *invariant* and *variant* expressions for loops. This feature is generally not obtainable for general programming languages, where iteration can loop over a user-defined data structure, e.g. a linked list. In such cases, advanced verification techniques such as abstract interpretation are needed for invariant inference [37].

5.7 Summary

In this chapter, we have encoded a sound execution semantics for ATL transformations, and adapted the VeriATL verifier for this task. We have explained precisely the runtime behaviour of ASM implementations by encoding a translational semantics of the ASM language in Boogie. We have also articulated a translation validation approach to verify the soundness of our Boogie encoding for the execution semantics of the ATL matched rule. Consequently, we are able to soundly verify the correctness of an ATL transformation against its specified OCL contracts. This is based on our sound encoding of the execution semantics of the ATL matched rules.

We have encapsulated the semantics of the EMF metamodel, OCL, and ASM as Boogie libraries in our VeriMTLr framework. Thus, we potentially facilitate verifier design for MTr languages other than ATL. However, we have not yet quantified how useful these libraries are in verifier design for model transformation languages from other transformation paradigms. In the next chapter, we will investigate the differences between the execution semantics of relational and graph transformations, and quantify how these differences affect verifier design. Moreover, we will demonstrate how to adapt the VeriMTLr framework to design a sound verifier which can be used to verify the correctness of graph transformations.

Chapter 6

A Modular and Sound Verifier Design for SimpleGT Graph Transformations

In Chapter 3 and Chapter 5 we have presented our design of several modular and reusable Boogie libraries. These libraries are encapsulated into our VerMTLr framework which facilitates systematic verifier construction for MTr languages such as ATL. In this chapter, we adapt VerMTLr to design the VeriGT verifier, which allows us to soundly prove the partial correctness of SimpleGT graph transformations. In Section 6.1, we present an overview of the VeriGT verifier, including the basic concepts of the SimpleGT language. The main contributions of this chapter are a semantics for the SimpleGT language and a semantics for EMFTVM bytecode. In particular, we will demonstrate the differences between the execution semantics of relational and graph transformations, and quantify how the differences would affect their verifier designs (Section 6.2). We will also illustrate how to develop the semantics of EMFTVM bytecode by extending the semantics of ASM bytecode (Section 6.3), to enable a translation validation approach for a wider range of model transformation languages (Section 6.4). In Section 6.5, we evaluate the VeriGT verifier on the well-known Pacman game, showing the performance and feasibility of VeriGT. Finally, we will discuss the design of VeriGT compared to other related GT verification techniques and tools.

6.1 Introduction to VeriGT

As already discussed, MTr is one of the main paradigms used in model transformation. It has a “mapping” style, and aims at producing a declarative transformation specification that documents what the model transformation intends to do. GT is another model transformation paradigm. It uses a rewriting style, and usually represents a model transformation graphically (e.g. UML-related models) and at a high level of abstraction. Thus, it is well suited to describe scenarios such as distributed systems or the behaviour of structure-changing systems (e.g. mobile networks). The two paradigms share some similarities (e.g. they are both declarative in nature). However, they are fundamentally different in their execution semantics.

SimpleGT is a experimental GT language based on double push-out semantics developed by Wagelaar et al. [118]. A SimpleGT program is a declarative specification that documents what the SimpleGT transformation intends to do. It is expressed in terms of a list of rewrite rules, using OCL for both its data types and its declarative expressions. Then, the SimpleGT program is compiled into an EMFTVM implementation to be executed.

EMFTVM is a stack-based virtual machine, which aims at providing a common execution semantics for the implementation of rule-based model transformation languages [118]. It is based on EMF (which represents a de facto standard for modelling today), and uses the EMFTVM language to implement model transformations. Existing model transformation languages that target the EMFTVM include ATL and SimpleGT [118].

Example 6.1.1. We use the Pacman game adapted from [112] to introduce the SimpleGT language. The game is based on the Pacman metamodel as shown in Fig. 6.1. The game consists of a single *Pacman*, a *ghost* and zero or more *gems* on a game board (consisting of more than zero *grids*). Each grid can hold Pacman, a ghost and a gem at the same time. The Pacman game is controlled by the *GameState*, which records important attributes such as *STATE*, *SCORE* and *FRAME*. It also contains a list of actions. Each action defines the moves to be done by either Pacman or the ghost, and is executed when it has the same frame as the GameState.

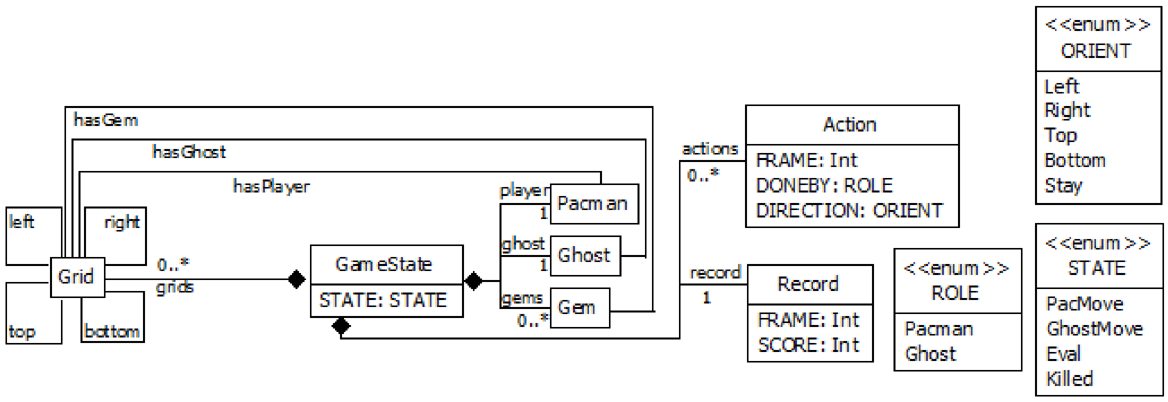


Fig. 6.1 Pacman metamodel

We have defined the semantics of a Pacman game via *13* GT rules in SimpleGT (Fig. 6.2). We have 10 rules to move Pacman and the ghost in different directions (5 rules for each role). We ensure that Pacman moves before the ghost. However, the evaluation (i.e. *Kill* or *Collect* rule) takes place after both of them have moved. Pacman collects a gem if both the gem and Pacman share the same grid. Pacman is killed by the ghost if both of them share the same grid. Finally, the GameState is updated by the *UpdateFrame* rule.

Each rule includes an input pattern (*from* section), a correspondence pattern, and an output pattern (*to* section). The correspondence pattern is implicit, and is represented by the intersection of the input and the output pattern. Thus, the coarse operational semantics of SimpleGT is that the difference between the input pattern and the correspondence pattern is deleted, the correspondence pattern is left unchanged, and the difference between the output pattern and the correspondence pattern is created. SimpleGT uses explicit Negative Application Conditions (NACs), which specify input patterns that prevent the rule from matching. Optionally, the matching operator ('=$\sim$') can be used to match the existence of an edge or an attribute value in the input or output pattern.

Take the *PlayerMoveLeft* rule of Fig. 6.2 for example; its input pattern specifies that:

- The game is in a state *s* when *Pacman* should move (line 5), and
- The *grid1* that contains *Pacman* has a left *grid2* beside it (line 6), and
- The *grid2* does not have the *ghost* on it (NAC, line 8), and
- An action *act* of move *left* that is to be performed by the *Pacman* at the current frame (line 7).

Then, the output pattern of the *PlayerMoveLeft* rule specifies that:

- The game is in a state that *Ghost* should move (line 10), and
- *Pacman* moves to the left of *grid1* (line 11).

The implicit correspondence graph of the *PlayerMoveLeft* rule is calculated as shown in Fig. 6.3. Thus, what have to be deleted (i.e. the difference between the input pattern and the correspondence pattern) are the value of *STATE* of game state *s*, the value of *hasPlayer* of *grid1*, the *act* and all value of its structural features. What have to be created (i.e. the difference between the output pattern and the correspondence pattern) are the value of *STATE* of game state *s* that becomes *GhostMove*, and the value of *hasPlayer* of *grid2* that sets to *Pacman*.

```
module Pacman;

rule PlayerMoveLeft{
  from
    s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
    grid2: P!Grid,grid1: P!Grid(hasPlayer=~pac, left=~grid2),
    act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Left)
  not grid2: P!Grid(hasEnemy=~ghost), ghost: P!Ghost
  to
    s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
    grid2: P!Grid(hasPlayer=~pac),grid1: P!Grid(left=~grid2)
}
... --# another 4 rules for Pacman to move in the other direction

rule ghostMoveLeft{
  from
    s:P!GameState(STATE=~GhostMove,record=~rec), rec: P!Record, ghost: P!Ghost,
    grid2:P!Grid, grid1: P!Grid(hasEnemy=~ghost, left=~grid2),
    act : P!Action(DONEBY=~Ghost, FRAME=~rec.FRAME, DIRECTION=~Left)
  to
    s:P!GameState(STATE=~Eval,record=~rec),rec: P!Record,ghost: P!Ghost,
    grid2: P!Grid(hasEnemy=~ghost), grid1: P!Grid(left=~grid2)
}
... --# another 4 rules for Ghost to move in the other direction

rule Collect{
  from
    s : P!GameState(STATE=~Eval,record=~rec),rec : P!Record,pac: P!Pacman,
    gem: P!Gem,grid : P!Grid(hasPlayer=~pac, hasGem=~gem)
  to
    s : P!GameState(STATE=~Eval,record=~recNew),grid : P!Grid(hasPlayer=~pac),
    pac: P!Pacman,recNew: P!Record(FRAME=~rec.FRAME, SCORE=~rec.SCORE+100)
}

rule Kill{
  from
    s : P!GameState(STATE=~Eval),ghost: P!Ghost,pac : P!Pacman,
    grid : P!Grid (hasPlayer=~pac, hasEnemy=~ghost)
  to
    s: P!GameState(STATE=~Killed),ghost: P!Ghost,grid : P!Grid (hasEnemy=~ghost)
}

rule UpdateFrame{
  from
    s : P!GameState(STATE=~Eval,record=~rec),rec : P!Record,pac : P!Pacman
  to
    s: P!GameState(STATE=~PacMove,record=~recNew),pac : P!Pacman,
    recNew: P!Record(FRAME=~rec.FRAME+1, SCORE=~rec.SCORE)
}
```

Fig. 6.2 Graph transformation rules for *Pacman* in SimpleGT

```
s : P!GameState(record=~rec),rec: P!Record,pac: P!Pacman,
grid2: P!Grid,grid1: P!Grid(left=~grid2)
```

Fig. 6.3 The correspondence graph of the *PlayerMoveLeft* rule

```
context Pacman!GameState pre ValidBoard: -- any two grids are reachable.
  Pacman!GameState->allInstances()->forAll(g | g.grids->forAll(g1,g2 | reachable(g1,g2)));

... -- other well-formedness contracts of the Pacman game.

context Pacman!Grid post gemReachable: -- all grids containing a gem must be reachable by Pacman.
  Pacman!Grid.allInstances()->forAll(g1,g2|not g1.hasPlayer.isOclUndefined()
    and not g2.hasGem.isOclUndefined() implies reachable(g1,g2));

context Pacman!GameState post PacmanSurvive: -- exists a path where the ghost never kills Pacman.
  Pacman!GameState->allInstances()->forAll(g |
   g.STATE=GhostMove implies g.grids->forAll(g1|
     g1.hasEnemy.oclIsKindOf(Pacman!Ghost) implies not g1.hasPlayer.oclIsKindOf(Pacman!Pacman)));

context Pacman!Action post PacmanMoved: -- the Pacman must move within a time interval I.
  let acts:Sequence(Pacman!Action) = Pacman!Action.allInstances()->select(a|
   a.DONEBY=Pacman and not a.Direction=Stay)->asSequence() in
      Integer.allInstances->forAll(i|
        0<=i<acts->size()-1 implies acts->at(i+1).FRAME-acts->at(i).FRAME<=I);
```

Fig. 6.4 OCL contracts for Pacman

In addition, unlike GT languages with explicit flow control (e.g. Henshin [4]), SimpleGT follows an automatic "fall-off" rule scheduling, i.e. if no match is found for a GT rule, it falls off to match the next rule.

The correctness of a SimpleGT program is specified using OCL contracts. In this work, we specify three contracts as shown in Fig. 6.4, i.e. *gemReachable*, *PacmanSurvive* and *PacmanMoved*. The rationale behind each specified contract is explained further in Section 6.5. In addition, we enforce a list of preconditions that should hold before executing the GT. This is to ensure the game starts in a valid game state. For example, to ensure that no grid is isolated on the game board, we require that any two grids are reachable (defined in Section 6.5).

VeriGT Verification System. We have designed the VeriGT verification system to enable sound automated verification of partial correctness for SimpleGT (Fig. 6.5). It accepts a source EMF metamodel, a set of specified OCL contracts and SimpleGT transformation. The output is a decision regarding the correctness of the SimpleGT transformation.

Specifically, VeriGT generates the corresponding Boogie code from its inputs using four code generators:

- The EMF metamodels generate Boogie types and constants using the *EMF2Boogie* code generator.
- The OCL transformation contracts produce Boogie contracts using the *OCL2Boogie* code generator.

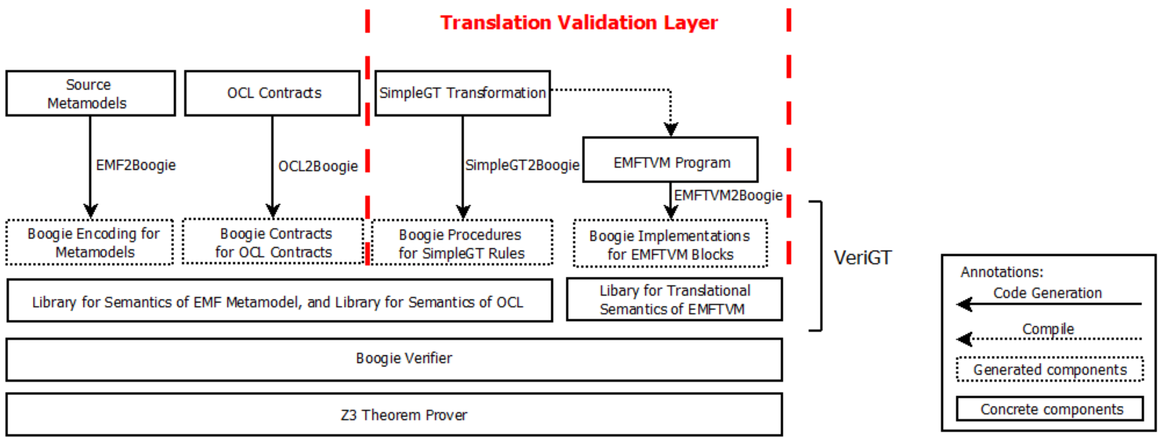

Translation Validation Layer
Source Metamodels
OCL Contracts
SimpleGT Transformation
EMF2Boogie
OCL2Boogie
SimpleGT2Boogie
EMFTVM Program
EMFTVM2Boogie
Boogie Encoding for Metamodels
Boogie Contracts for OCL Contracts
Boogie Procedures for SimpleGT Rules
Boogie Implementations for EMFTVM Blocks
Library for Semantics of EMF Metamodel, and Library for Semantics of OCL
Libary for Translational Semantics of EMFTVM
VeriGT
Boogie Verifier
Z3 Theorem Prover
Annotations:
Code Generation
Compile
Generated components
Concrete components

Fig. 6.5 Overview of the VeriGT verification system

- The SimpleGT transformation generates Boogie procedures using the *SimpleGT2Boogie* code generator.
- The SimpleGT compiler is used to compile input SimpleGT transformation into an EMFTVM program. This compiled EMFTVM program is used to generate a Boogie implementation using the *EMFTVM2Boogie* code generator.

Then, VeriGT performs translation validation to ensure the soundness of the Boogie procedure encoding for the execution semantics of SimpleGT. If the soundness verification passes, the generated Boogie procedures are thus verified against the generated Boogie contracts in OCL for transformation correctness verification. If the correctness of the SimpleGT transformation has been established, then VeriGT simply reports that the verification is successful. Otherwise, the trace information from the Boogie verifier, indicating where the transformation incorrectness was detected, will be output.

The generated Boogie code is driven by four core components of VeriGT, that is the semantics of the EMF metamodel, OCL, SimpleGT and EMFTVM. The first two are directly adopted from the VeriMTLr framework. These are also the foundation of the semantics of SimpleGT, which was one of the main challenges in developing VeriGT. Another challenge stems from the development of translational semantics of EMFTVM, which is the core component to enable translation validation of the soundness of our Boogie encoding. This semantics is an extension of our previous semantics of ASM (Section 5.2). In what follows, we first illustrate the semantics of SimpleGT.

6.2 Semantics of SimpleGT

In this section, we describe the semantics of rule scheduling as well as the semantics of *match* and *apply* steps for SimpleGT. In addition, we compare them with the semantics for MTr languages to quantify how their semantic differences affect the verifier design of SimpleGT (i.e. preventing us from reusing the Boogie libraries for the execution semantics of MTr to design VeriGT).

6.2.1 Semantics of Rule Scheduling

The semantics of rule scheduling in SimpleGT requires that to be able to match rules with their own output, i.e. re-matching after each apply[1]:

- Initially, rules are matched to find the source graph pattern as specified in the *from* section of the rule (*match step*).
- Next, the first match is applied, i.e. deleting input elements, creating output elements, and initializing output elements as specified in the *to* section of the rule (*apply step*).
- After each application, the rule scheduling restarts immediately.
- When all rules have been processed (i.e. there are no more matches found for any rules), the rule scheduling stops.

The rule scheduling of SimpleGT implies that the source and target models are the same. Thus, there will be only one *heap* in our Boogie encoding.

To the best of our knowledge, none of MTr languages share the same rule scheduling. Taking the ATL language as an example, ATL is scheduled to first match each rule, and then apply each rule. This is to ensure the confluence of an ATL transformation [6].

Example 6.2.1. In Fig. 6.6 we show part of the Boogie encoding for the execution semantics of the *Pacman* game. This can be verified against the OCL contracts specified in Fig. 6.4 as follows:

- First, the OCL contracts are encoded as a Boogie contract (line 2 - 16). For instance, the contract *PacmanSurvive* of Fig. 6.4 is encoded as both a precondition (line 2 - 7) and a postcondition (line 10 - 15).

[1]For simplicity we do not consider rule inheritance [118].

- Then, the execution semantics of the Pacman SimpleGT program is encoded as a Boogie implementation (line 17 - 33). Specifically, the rule scheduling is encoded in a loop (line 19 - 31). During the loop, the execution semantics of the *match* and *apply* steps of each rule are performed. If no match is found for a GT rule, it falls off to match the next rule (line 28). This fall-off-matching is repeated until the last GT rule exits the loop.

- Finally, we pair the Boogie contract that represents the specified OCL contracts with the Boogie implementation that represents the execution semantics of the SimpleGT program. Such a pair forms a verification task, which is input to the Boogie verifier. The Boogie verifier either gives a confirmation that indicates the SimpleGT program satisfies the specified OCL contracts, or trace information that indicates where the OCL contract violation is detected.

6.2.2 Semantics of the Match Step

The semantics of the *match step* of each SimpleGT rule consists of two sub-steps:

- The first sub-step performs a structural pattern matching (by applying a search plan strategy [115]), where all the patterns that match the specified model elements and their structural relationship (i.e. an edge between model elements) are found. A subtlety here is that SimpleGT requires injective matching, i.e. all the model elements in each matched structural pattern are unique.

- The second sub-step is to iterate on the matched structural patterns for semantic pattern matching, where a pattern that satisfies specified semantic constraints is found (i.e. the constraints on the attributes of model elements given by the matching operator, and any NACs).

To the best of our knowledge, none of the MTr languages implement the same matching algorithm. For example, ATL applies an implicit resolution algorithm while binding the target metamodel elements, which does not exist in SimpleGT. In addition, ATL does not enforce injective matching.

Therefore, the encoded Boogie contract for the *match* step has the following structure:

- It ensures that if the result is an empty sequence, then the source model does not contain any pattern that passes the structural and semantic pattern matching.

```
procedure main();
/* inv: PacmanSurvive */
requires (∀ gs1: ref • (gs1∈OCLType#allInstance(srcHeap, pacman$GameState)
  ∧ read(srcHeap, gs1, pacman$GameState.STATE)=4) ⟹
    (∀ grid1: ref • grid1∈OCLType#allInstance(srcHeap, pacman$Grid)
        ∧ dtype(read(srcHeap, grid1 , pacman$Grid.hasEnemy))<: pacman$Ghost ⟹
      ¬(dtype(read(srcHeap, grid1 , pacman$Grid.hasPlayer))<: pacman$Pacman) ));
...
modifies srcHeap;
/* inv: PacmanSurvive */
ensures (∀ gs1: ref • (gs1∈OCLType#allInstance(srcHeap, pacman$GameState)
  ∧ read(srcHeap, gs1, pacman$GameState.STATE)=4) ⟹
    (∀ grid1: ref • grid1∈OCLType#allInstance(srcHeap, pacman$Grid)
      ∧ dtype(read(srcHeap, grid1 , pacman$Grid.hasEnemy))<: pacman$Ghost ⟹
       ¬(dtype(read(srcHeap, grid1 , pacman$Grid.hasPlayer))<: pacman$Pacman) ));
...
implementation main() {
  ... // variable declarations
  while(true) ... {
    Label_PlayerMoveLeft:
      Label_Match_PlayerMoveLeft:
        call p:=match_PlayerMoveLeft();
      Label_Apply_PlayerMoveLeft:
      if(p≠ []){
        call apply_PlayerMoveLeft(p);
        goto Label_restart;}
      else{ goto Label_PlayerMoverRight; }
    Label_PlayerMoverRight:
      ...
    Label_restart:
  }
  Label_exit_point:
}

procedure match_PlayerMoveLeft() returns (res: Seq ref);
... // Boogie contract for the execution semantics of match step.
procedure apply_PlayerMoveLeft(res: Seq ref);
... // Boogie contract for the execution semantics of apply step.
```

Fig. 6.6 Boogie encoding to verify the correctness of the *Pacman* transformation

- It ensures that if the result is not an empty sequence, then the result is a pattern (contains a sequence of source elements) in the source model that passes the structural and semantic pattern matching.

Example 6.2.2. The Boogie encoding for the *match* step for the *PlayerMoveLeft* rule is shown in Fig. 6.7. The encoding is self-explanatory and conforms to the structure of the Boogie contract for the *match* step:

- If the result is an empty sequence, then in the source model, none of the pattern that passes the structural pattern matching (specified by the *findPatterns_PlayerMoveLeft* function) satisfies the constraints of semantics pattern matching (line 2 - 15):
  - The game is in a state *s* when *Pacman* should move (line 5).

  - The action *act* is done by *Pacman* (line 7).
  - The action *act* has the same *FRAME* as the game *FRAME* (line 9 - 10).
  - The action *act* is to move to the *LEFT* (line 12).
  - The grid *grid2* does not have the *ghost* on it (line 14 - 15).

- If the result is not an empty sequence, then the result is a pattern (contains a sequence of source elements) in the source model that passes the structural and semantic pattern matching (line 16 - 23).

```
1  procedure match_PlayerMoveLeft() returns (res: Seq ref)
2  ensures res=[] ⟹
3    (∀ i: int • 0≤i<|findPatterns_PlayerMoveLeft(srcHeap)|) ⟹ ¬(
4    // s : pacman$GameState(STATE=~PacMove)
5    read(srcHeap,findPatterns_PlayerMoveLeft(srcHeap)[i][0],pacman$GameState.STATE) = 3
6    // act : pacman$Action(DONEBY=~Pacman)
7    ∧ read(srcHeap,findPatterns_PlayerMoveLeft(srcHeap)[i][5],pacman$Action.DONEBY) = 1
8    // act : pacman$Action(FRAME=~rec.FRAME)
9    ∧ read(srcHeap,findPatterns_PlayerMoveLeft(srcHeap)[i][5],pacman$Action.FRAME) =
10     read(srcHeap, findPatterns_PlayerMoveLeft(srcHeap)[i][1], pacman$Record.FRAME)
11   // act : pacman$Action(DIRECTION=~Left)
12   ∧ read(srcHeap,findPatterns_PlayerMoveLeft(srcHeap)[i][5],pacman$Action.DIRECTION)=1
13   // not grid2: pacman$Grid(hasEnemy=~ghost), ghost: P$Ghost
14   ∧ ¬(dtype(read(srcHeap,findPatterns_PlayerMoveLeft(srcHeap)[i][3],pacman$Grid.hasEnemy))
15     <: pacman$Ghost)));
16 ensures res ≠ [] ⟹
17   res ∈ findPatterns_PlayerMoveLeft(srcHeap)
18    ∧ read(srcHeap,res[0],pacman$GameState.STATE) = 3
19    ∧ read(srcHeap,res[5],pacman$Action.DONEBY) = 1
20    ∧ read(srcHeap,res[5],pacman$Action.FRAME) =
21       read(srcHeap, res[1], pacman$Record.FRAME)
22    ∧ read(srcHeap,res[5],pacman$Action.DIRECTION)=1
23    ∧ ¬(dtype(read(srcHeap, res[3], pacman$Grid.hasEnemy))<: pacman$Ghost);
24
25 function findPatterns_PlayerMoveLeft(): Seq< Seq<ref> >;
26 ... // Boogie axioms for the structural pattern matching.
```

Fig. 6.7 Boogie encoding for the *match* step for the *PlayerMoveLeft* rule

6.2.3 Semantics of the Apply Step

The semantics of the *apply step* of each SimpleGT rule is more straightforward than that of the *match step*. One caveat here is that SimpleGT is a programming language with explicit memory deallocation (e.g. delete model element). When this occurs, the frame condition, that each model element allocated before executing the *apply* step is still allocated, no longer holds.

Another aspect of the semantics of the *apply* step is that we introduce a new Boogie type *setTable* to prohibit runtime exceptions caused by certain operations on the structural features, e.g. adding the same attribute twice for a model element:

```
type setTable = <α>[ref, Field α] bool;
var acc: setTable;
```

Thus, checking whether *o.f* is set or not becomes an expression *isset*(*acc*,o,f), marking *o.f* as set uses the expression *set*(*acc*,o,f,*true*), and marking it as not set uses the expression *set*(*acc*,o,f,*false*).

The encoded Boogie contract for the *apply* step has the following structure:

- It requires that the received pattern passes the structural and semantic pattern matching of its corresponding SimpleGT rule.
- It guarantees that the structural features of the received pattern, that are to be accessed by the *apply* step, are all set.
- It specifies that the heap for the source model and *setTable* will be modified.
- It ensures that each target element is fully applied, i.e. deleting elements, creating elements, and initializing elements as specified in the corresponding SimpleGT rule.
- It ensures the structural features for the deleted, and initialised model elements are set/unset as specified in the corresponding SimpleGT rule.
- It addresses the frame problem by ensuring that nothing else is modified on the source heap, except the specified application performed on each model element.
- It addresses the frame problem by ensuring that nothing else is modified in the *setTable*, except the structural features of the affected model elements.

Notice that we assume the structural features to be accessed from the received input pattern are all set before executing the *apply* step, since it passes the structural and semantic pattern matching. That is, the source elements' structural features have been set in order to be accessed. The NACs are an exception, because SimpleGT could use NAC to check whether the structural features of specified source elements are set or not (which means source elements' structural features do not have to be set to be accessed). In addition, the state of the *setTable* needs to be updated and propagated by the postconditions and the frame condition.

Example 6.2.3. The Boogie encoding for the *apply* step of the *PlayerMoveLeft* rule is shown in Fig. 6.8:

- First, it requires that the input pattern passes the structural and semantic pattern matching of the *PlayerMoveLeft* rule (line 2 - 9).
- It also guarantees that the to-be-accessed structural features of the input pattern are all set (line 10 - 15).
- Then, it specifies that the heap for the source model, and *setTable* will be modified (line 17).
- Next, it ensures that each element is fully applied, by performing associated model manipulations as specified by the *PlayerMoveLeft* rule (line 19 - 26).
- It ensures the structural features for the initialised model elements are set/unset as specified in the corresponding SimpleGT rule (line 28 - 33), and the structural features for the deleted model element are unset (line 34 - 39).
- It also ensures that nothing else is modified in the input model, except the model manipulations performed on the specified model element (line 41 - 47).
- Finally, it ensures that nothing else is modified in the *setTable*, except the affected structural features for the deleted and initialised elements (line 48 - 53).

The Boogie contracts for the execution semantics of the *match* and *apply steps* play an important role in verifying the correctness of a SimpleGT Transformation. Thus, the soundness of our approach depends on the soundness of these Boogie contracts, i.e. that they correctly represent the runtime behaviour of their corresponding EMFTVM implementations. In the next sections, we describe our translation validation approach to verify the soundness of our Boogie encodings for the execution semantics of SimpleGT.

6.3 Translational Semantics of the EMFTVM Language

Our translation validation approach is based on providing a translational semantics of the EMFTVM language in Boogie, which allows the runtime behaviour of the EMFTVM implementation to be represented using Boogie implementations.

```
procedure apply_PlayerMoveLeft(res: Seq ref);
// input pattern passes the structural and semantic pattern matching.
requires res ∈ findPatterns_PlayerMoveLeft(srcHeap)
   ∧ read(srcHeap,res[0],pacman$GameState.STATE) = 3
   ∧ read(srcHeap,res[5],pacman$Action.DONEBY) = 1
   ∧ read(srcHeap,res[5],pacman$Action.FRAME) =
       read(srcHeap, res[1], pacman$Record.FRAME)
   ∧ read(srcHeap,res[5],pacman$Action.DIRECTION)=1
   ∧ ¬(dtype(read(srcHeap,res[3], pacman$Grid.hasEnemy))<: pacman$Ghost);
// to-be-accessed structural features of the input pattern are all set.
free requires isset(acc,res[0],pacman$GameState.STATE);
free requires isset(acc,res[1],pacman$Record.FRAME);
free requires isset(acc,res[5],pacman$Action.DONEBY);
free requires isset(acc,res[5],pacman$Action.FRAME);
free requires isset(acc,res[5],pacman$Action.DIRECTION);

modifies srcHeap, acc;

// s : pacman$GameState(STATE=~4)
ensures read(srcHeap,res[0],pacman$GameState.STATE) = 4;
// grid2: pacman$Grid(hasPlayer=~pac)
ensures read(srcHeap,res[3],pacman$Grid.hasPlayer) = res[2];
// grid1: pacman$Grid(hasPlayer=~null)
ensures read(srcHeap,res[4],pacman$Grid.hasPlayer) = null;
// act : pacman$Action(alloc=~false)
ensures ¬read(srcHeap,res[5],alloc);

// pacman$GameState.STATE isset for s.
ensures isset(acc,res[0],pacman$GameState.STATE);
// pacman$Grid.hasPlayer isset for grid2.
ensures isset(acc,res[3],pacman$Grid.hasPlayer);
// pacman$Grid.hasPlayer unset for grid1.
ensures ¬isset(acc,res[4],pacman$Grid.hasPlayer);
// pacman$Action.DONEBY unset for act.
ensures ¬isset(acc,res[5],pacman$Action.DONEBY);
// pacman$Action.FRAME unset for act.
ensures ¬isset(acc,res[5],pacman$Action.FRAME);
// pacman$Action.DIRECTION unset for act.
ensures ¬isset(acc,res[5],pacman$Action.DIRECTION);

// Frame condition for input model.
ensures (∀<α> o: ref, f: Field α •
  o ≠ null ∧ read(old(srcHeap),o,alloc) ⟹
    (dtype(o)=pacman$GameState ∧ f=pacman$GameState.STATE)
    ∨(dtype(o)=pacman$Grid ∧ f=pacman$Grid.hasPlayer)
    ∨(dtype(o)=pacman$Action ∧ f=alloc)
    ∨(read(srcHeap,o,f)=read(old(srcHeap),o,f)));
// Frame condition for setTable.
ensures (∀<α> o: ref, f: Field α •
    (o=res[0] ∧ f=pacman$GameState.STATE)
    ∨((o=res[3]∨o=res[4]) ∧ f=pacman$Grid.hasPlayer)
    ∨(o=res[5] ∧ (f=pacman$Action.DONEBY∨f=pacman$Action.FRAME∨f=pacman$Action.DIRECTION))
    ∨(read(acc,o,f)=read(old(acc),o,f)));
```

Fig. 6.8 Boogie encoding for the *apply* step for the *PlayerMoveLeft* rule

The EMFTVM language supports 47 different instructions, which forms a superset of the ASM language. Compared to ASM, the EMFTVM language adds more control-flow instructions to model more flexible runtime execution flow. In addition, it increases the number of model-handling instructions that are dedicated to model manipulation (e.g. DELETE, ADD, INSERT, REMOVE). Since SimpleGT allows explicit memory deallocation, these additional EMFTVM instructions are extensively used in its EMFTVM implementation.

Our translational semantics for EMFTVM is given by a list of translation rules to Boogie. Each translation rule encodes the operational semantics of an instruction in Boogie. Our translational semantics of the EMFTVM language extends our translational semantics of the ASM language. While developing the translational semantics of the EMFTVM language, we have not found documents that precisely explain the operational semantics of EMFTVM bytecode instructions. Our strategy is to check the EMFTVM source code for the operational semantics of each EMFTVM instruction, and then design the rule correspondingly.

Specifically, 17 instructions are shared by both the EMFTVM and ASM languages. In these cases, translational semantics are inherited. In addition, we add translational semantics for 9 EMFTVM instructions. The remaining 21 instructions, whose translation semantics are listed in Appendix D, are not used in the EMFTVM implementation of the SimpleGT language. The translational semantics of the EMFTVM language (the part that is extended from the ASM language) is shown in Table 6.1. It is encapsulated as a Boogie library, which can be adapted by the verifier designer for other model transformation languages.

In what follows, we pick two representative EMFTVM instructions as our examples, i.e. the *DELETE* and *ADD* instructions. We first give an informal description of their operational semantics, and then explain the intuition behind the corresponding translation to Boogie whenever it is necessary.

The *DELETE* instruction is one of the EMFTVM instructions for model handling. The *DELETE* instruction does not have any parameters. Before executing the *DELETE* instruction, the top element on the operand stack is a model element o.

The operational semantics of the *DELETE* instruction simply deletes the element from the source model. Two subtleties here are that:

- If the deleted element is not a collection, then the cross-references between the deleted element and any other element in the source model are also deleted.
- If the deleted element is a collection, then all its sub-elements are moved outside the source model.

Finally, the top element on the operand stack is popped.

The translation rule for the *DELETE* instruction requires some explanation in order to describe our approach to modelling element deallocation:

- First, we prevent null or unallocated model element from being deallocated.

ASM Instruction (S)	**Corresponding Boogie Statements ([[S]])**
Stack Handling Instructions	
AND *Stmt*	`assert` size(Stk)>0 ; [[*Stmt*]]; Stk := hd(tl(stk)) $\wedge$ hd(stk) :: tl(tl(Stk));
NOT	`assert` size(Stk)>0 ; Stk := !(hd(stk)) :: tl(Stk);
Control Flow Instructions	
IFN *n*	`var` $\text{cond}^{\#}$: bool ; `assert` size(Stk) $>$ 0 ; $\text{cond}^{\#}$:= hd(Stk); Stk := tl(Stk) ; `if` (!$\text{cond}^{\#}$) `goto` *l* ; (where *l* is a fresh label. It labels the program point which corresponds to the EMFTVM instruction offset *n*)
RETURN	`goto` *END*;
Model Handling Instructions	
ALLINST *cl*	`var` $\text{col}^{\#}$: Seq ref ; `assert` size(Stk) $>$ 0 ; $\text{col}^{\#}$:= OCLType#allInstance(heap, cl); Stk := $\text{col}^{\#}$::tl(Stk) ;

DELETE	`let` o=hd(Stk) `in`
	`assert` size(stk) >= 1;
	`assert` o != null $\wedge$ read(heap, o, alloc);
	`havoc` heap;
	`assume` ($\forall$ r: ref, f: Field α ::
	r!=null $\wedge$ read(heap', r, alloc) $\wedge$ r!=o ==>
	read(heap, r, f) == read(heap', r, f));
	`assume` $\neg$ read(heap, o, alloc);
	Stk := tl(Stk);

ADD *f*	let o = hd(tl(Stk)), v = hd(Stk) in `assert` size(Stk) $>$ 1 $\wedge$ $o \neq null \ \wedge \ read(heap, o, alloc)$; `if` (isCollection(f)) { heap := update(heap, read(heap, o, f), read(heap, o, f) $\cup$ v); } `else` { `assert` !isset(acc, o, f); heap := update(heap, o, f, v); acc := set(acc, o, f, true); } Stk := tl(tl(Stk));
REMOVE *f*	let o = hd(tl(Stk)), v = hd(Stk) in `assert` size(Stk) $>$ 1 $\wedge$ $o \neq null \ \wedge \ read(heap, o, alloc)$; `if` (isCollection(f)) { heap := update(heap, read(heap, o, f), read(heap, o, f) - v) ; } `else` { `if`(read(heap, o, f)==v) { `assert` isset(acc, o, f);heap := update(heap, o, f, default);acc := set(acc, o, f, false); } } Stk := tl(tl(Stk));
INSERT *f*	let o = hd(tl(tl(Stk))), v = hd(tl(Stk)), i=hd(Stk) in `assert` size(Stk) $>$ 1 $\wedge$ $o \neq null \ \wedge \ read(heap, o, alloc)$; `if` (isCollection(f)) { `assert` -1<=i<size(read(heap, o, f)); heap:=update(heap, o, f, read(heap, o, f)[0..i] :: v ::read(heap, o, f)[i..size(read(heap, o, f))]); } `else` { `assert` !isset(acc, o, f); heap := update(heap, o, f, v); acc := set(acc, o, f, true); } Stk := tl(tl(tl(Stk)));

Table 6.1 Translational semantics of the EMFTVM language

- Second, we specify that the heaps before (denoted by *heap'*) and after (denoted by *heap*) calling the *DELETE* instruction agree on the values of all the allocated model element, except the element to be deallocated. Consequently, the values of deallocated model elements are "forgotten" after calling the *DELETE* instruction.
- Third, we arrange to set the implicit allocation field of the deallocated element to *false*. The implication is that it will no longer be considered when quantifying over allocated model elements.

As a result, our translation rule for the *DELETE* instruction subsumes the subtleties in its operational semantics:

- If the deleted element is not a collection, then any other elements in the source model that reference the deleted element will reference an inaccessible element.
- If the element is a collection, then all its sub-elements are no longer accessible after deletion (i.e. not contained by the source model), since accessing the sub-elements of an unallocated model element is always prohibited by our Boogie encoding.

The *ADD* instruction is another EMFTVM instruction for model handling. The *ADD* instruction has one parameter which is the structural feature to be operated on. Before executing the *ADD* instruction, the top two elements on the operand stack are a model element *o* and the value *v* (to add to *o*).

The operational semantics of the *ADD* instruction forms a case distinction according to its parameter *f*. If *f* is an association and its multiplicity has an upper-bound that is greater than one, then the *ADD* and *SET* instructions agree on their behaviour. Otherwise, *o.f* is checked for whether it has already been set to any value. If *o.f* has already been set to a value, a runtime exception is thrown. Otherwise, the *ADD* instruction successfully updates *o.f* to *v*, and marks *o.f* as being set. Finally, the top two elements on the operand stack are popped.

The translation rule for the *ADD* instruction offers no surprise based on its operational semantics.

6.4 Translation Validation of Encoding Soundness

Each SimpleGT rule is actually compiled into two EMFTVM implementation blocks by the EMFTVM compiler, i.e. a *match block* (for the *match step*) and an *apply block* (for the *apply step*). In this section, we briefly describe our translation validation approach to verify the soundness of our Boogie encodings for the execution semantics of SimpleGT, i.e. that our Boogie encoding for the execution semantics of SimpleGT soundly represents the corresponding runtime behaviour given by the EMFTVM implementation.

In order to verify the soundness of our Boogie encoding for the execution semantics of each SimpleGT rule, we define the execution semantics of an EMFTVM rule encoded in Boogie as being sound, if,

- the Boogie contract that represents the execution semantics of its *match step* is satisfied by the Boogie implementation that represents the runtime behaviour of its *match block*, and
- the Boogie contract that represents the execution semantics of its *apply step* is satisfied by the Boogie implementation that represents the runtime behaviour of its *apply block*.

Each of these conditions form a verification task that is sent to the Boogie verifier. If none of the verification tasks generate any errors (from the verifier), we conclude that our Boogie encoding for the execution semantics of the SimpleGT rules is sound. The benefit is that we can automatically verify the soundness of each SimpleGT specification/EMFTVM implementation pair.

We demonstrate our approach on the *match step* of the *PlayerMoveLeft* rule (Fig. 6.9). The runtime behaviour of its corresponding *match block* is implemented as a Boogie implementation (Line 5 - 12). The implementation contains two important sub-steps.

The first sub-step performs a structural pattern matching (Line 8), where all the patterns that match the specified model elements and their structural relationships (i.e. edges between model elements) are found. Structural pattern matching is mainly implemented in Java instead of EMFTVM. Thus, they are only axiomatised using Boogie axioms (Line 15), and are not validated by the translation validation approach. This is a modular verification strategy, thus initiating the verification of the implementation for the SimpleGT transformations: we aim to verify each sub-component of the transformation implementation individually. Consequently, we can clearly state which parts of the implementation are verified, which brings us a step closer to fully verifying the transformation implementation.

The second sub-step is to iterate on the matched structural patterns (Line 9 - 12) for semantic pattern matching, where a pattern that satisfies the specified semantic constraint is found (i.e. the constraint on attributes of model elements given by the matching operator). The runtime behaviour of semantic pattern matching is given as the Boogie implementation (Line 24 - 31) in terms of the translational semantics of EMFTVM. It is validated against

```
1  procedure match_PlayerMoveLeft() returns (p: Seq ref);
2  ... /* Boogie contract for the execution semantics of match step. */
3
4
5  /* Boogie implementation for the execution semantics of match step. */
6  implementation match_PlayerMoveLeft() returns (p: Seq ref);
7  { ... p:=[];i:=0;
8    patterns:=findPatterns_PlayerMoveLeft();
9    while(i<patterns.Length)...{
10   call matched:=match_filter_PlayerMoveLeft(patterns[i]);
11   if(matched){ p=patterns[i]; break;}
12   i:=i+1;} }
13
14 function findPatterns_PlayerMoveLeft(): Seq< Seq<ref> >;
15 ... // Boogie axioms for the structural pattern matching.
16
17 procedure match_filter_PlayerMoveLeft(p: Seq ref) returns (r: bool);
18 /* Boogie contract for the semantic pattern matching. */
19  ensures r ⟺ ( read(srcHeap,p[0],pacman$GameState.STATE) = 3
20    ∧ read(srcHeap,p[5],pacman$Action.DONEBY) = 1
21    ∧ read(srcHeap,p[5],pacman$Action.FRAME)=read(srcHeap, p[1], pacman$Record.FRAME)
22    ∧ read(srcHeap,p[5],pacman$Action.DIRECTION)=1);
23
24 /* Boogie implementation for the semantic pattern matching. */
25 implementation match_filter_PlayerMoveLeft(p: Seq ref) returns (r: bool)
26 { ... s,rec,pac,grid2,grid1,act:=p[0],p[1],p[2],p[3],p[4],p[5];
27   call stk:= init(); /* init local stack */
28   call stk := OpCode#Load(stk, s); /* load (s: pacman$GameState) */
29   call stk := OpCode#Get(stk, pacman$GameState.STATE); /* get STATE */
30   call stk := OpCode#Push(stk, 3);  /* push 3 */
31   call b:= Native#MatchOperator();  /* invoke (opName: =~) */ ... }
```

Fig. 6.9 Verification of the soundness of Boogie encodings for the *match step* of the *PlayerMoveLeft* rule

the Boogie contracts for semantic pattern matching (Line 17 - 22) to ensure its encoding soundness.

The verification of the soundness of the Boogie encodings for the *apply* step is performed in a similar way to what is done for the *match* step.

Finally, we can conclude that the execution semantics of a SimpleGT specification encoded in Boogie is sound when the execution semantics of both *match* and *apply* steps of all the relevant SimpleGT rules encoded in Boogie are sound.

6.5 Evaluation of VeriGT

VeriGT contains four code generators to generate Boogie code in order to soundly prove the partial correctness of SimpleGT graph transformations. Two code generators, *EMF2Boogie* and *OCL2Boogie*, are reused to produce their corresponding Boogie code for EMF metamodels and OCL transformation contracts (Section 4.3). The code generator of *SimpleGT-*

2Boogie and *EMFTVM2Boogie* have not yet being implemented. Thus, at the time of this thesis being written, their corresponding Boogie code is manually encoded to demonstrate the feasibility of VeriGT. However, we anticipate that their implementations are similar to what we described in Section 4.3 and Section 5.4 using a mixture of model-to-text technology and Java.

We evaluate VeriGT on the Pacman transformation, previously presented by Syriani and Vangheluwe [112]. The Pacman transformation, as shown in Fig. 6.2, gives the operational semantics of the Pacman game. Although the transformation is simple, it contains all the language features that VeriGT supports for the SimpleGT language, i.e. double push-out semantics with NAC, injective matching, and automatic "fall-off" rule scheduling. Table 6.2 summarises its verification complexity in terms of *7* metrics. The full case study, consisting of the involved metamodels, the specified OCL transformation contracts, and the SimpleGT transformation, can be found in Appendix C. Moreover, we refer to our online repository for the generated Boogie programs of the Pacman transformation in SimpleGT [31].

Metric	**Pacman**
No. Classifiers (source mm)	7
No. Attributes (source mm)	6
No. Associations (source mm)	13
No. ATL rules	13
No. ATL rule filter	40
No. OCL contracts (total/pre/post)	10/7/3
Complexity of OCL contract (total/average)	75/8

Table 6.2 The verification complexity metrics for the Pacman transformation

Our evaluation uses the Boogie verifier (version 2.2) and Z3 (version 4.3). It runs on an Intel 2.93 GHz machine with 4 GB of memory running on the Windows operation system. Verification times are recorded in seconds.

Table 6.3 shows the performance when automatically verifying the soundness of our Boogie encoding for the *Pacman* transformation. The *second* and *third* columns show the size of the Boogie code generated for the *match* and *apply* step of each SimpleGT rule respectively (shown by lines of Boogie contract/Boogie implementation). Their corresponding verification time is shown in the *fourth* and *fifth* columns. We report that all the verification tasks are verified automatically.

Table 6.4 shows the performance of the verification of the correctness of the Pacman transformation. The *second* column shows the size of the Boogie code generated for verifying each of the transformation contracts that are specified in Fig. 6.4 (including Boogie

Rule	**Boogie (LoC)**		**Veri. Time (s)**		**Automation**
	match	**apply**	**match**	**apply**	
PlayerMoveLeft	20/262	28/120	0.109	0.499	Auto
PlayerMoveRight	20/262	28/120	0.109	0.499	Auto
PlayerMoveTop	20/262	28/120	0.109	0.499	Auto
PlayerMoveBottom	20/262	28/120	0.109	0.499	Auto
PlayerStay	19/180	20/86	0.062	0.250	Auto
GhostMoveLeft	17/189	27/109	0.078	0.452	Auto
GhostMoveRight	17/189	27/109	0.078	0.452	Auto
GhostMoveTop	17/189	27/109	0.078	0.452	Auto
GhostMoveBottom	17/189	27/109	0.078	0.452	Auto
GhostStay	17/180	20/86	0.062	0.250	Auto
Collect	9/92	29/119	0.046	0.234	Auto
Kill	9/86	21/57	0.032	0.140	Auto
UpdateFrame	9/79	25/115	0.031	0.234	Auto
Total	211/2421	335/1379	0.981	4.912	

Table 6.3 Performance measures for verifying the soundness of the encoding for the Pacman transformation

encodings for the Pacman metamodel, transformation contract and rule scheduling). Corresponding verification times are shown in the *third* column.

	Boogie (LoC)	**Veri. Time (s)**
gemReachable	421	0.998
PacmanSurvive	467	1.747
PacmanMoved	439	0.109
Total	1327	2.854

Table 6.4 Performance measures for verifying the correctness of the Pacman transformation

Some explanation is in order. The first contract which we verified (*gemReachable*) was that all grid nodes containing a gem must be reachable by the Pacman. The key to this task is to define the *reachable* relation on grids. We define two grids to be reachable if they are adjacent to each other. The *reachable* relation is also reflexive, symmetric and transitive. Recall that to ensure no grid is isolated on the game board, we require that any two grids are reachable (including all the grids that containing the gem, or Pacman) as a precondition of the Pacman game. Since there are no rules that modify the layout of the grid, the first contract can be automatically verified with ease.

The second contract which we verified (*PacmanSurvive*) is that there exists a path where the ghost never kills Pacman. Our verification strategy is to provide a witnesses for the existence of such a path. First, we consider the state when the ghost starts to move. Then, under such a state, our goal is to verify that the ghost and Pacman do not share the same grid. Thus, the path where the ghost stays at the Pacman-free grid is our witness (recall that the move strategy of Pacman is not to commit suicide as shown in Fig. 6.2).

The third contract which we verified (*PacmanMoved*) is that Pacman must move within a time interval *I*. We use a contract-only variable (also known as a model field or a ghost variable [82]) for this task. Contract-only variables do not participate in the runtime execution of a program, as they are simply used to make the contract easier to express. In particular, we introduce the contract-only variable *acts*, which is a set of actions of Pacman that contains moves in any direction (corresponding to line 16 - 17 of Fig. 6.4). We need to explicitly update the contract-only variable *acts* when the action of Pacman is updated (e.g. delete an action as in the PlayerMoveLeft rule in Fig. 6.2), since it is not part of the runtime execution of a SimpleGT program. After that, we can automatically verify the third contract. This is due to the fact that if we assume that all the actions in *acts* will perform within a time interval *I* as a precondition of the Pacman game, then after we remove an action from *acts*, the remaining actions should not be changed and they will still be performed within a time interval *I*.

6.6 Analysis of VeriGT

GT verification is an active research area [1, 2]. In this section, we will compare the design of VeriGT to related GT verification techniques and tools.

Cabot et al. translate the rules of a GT in TGG into a set of OCL invariants that encode the transformation's behaviour. These invariants can be used to check properties (e.g. syntactic constraints) of the GT by applying bounded verification using the UML2CSP verifier [24]. Syriani and Vangheluwe propose an input-driven simulation approach using the Discrete EVent system Specification (DEVS) formalism [112]. Bill et al. extend OCL with Computational Tree Logic (CTL)-based temporal operators to express properties over the lifetime of a graph [16]. Both of these approaches are bounded, which means the GT is verified against its contracts within a given search space (i.e. using finite ranges for the number of models, associations and attribute values). Bounded approaches are usually automatic, but no conclusion can be drawn outside the search space. Our approach is based on automatic theorem proving, which is unbounded to ensure the contracts hold for the GT over an infinite domain. However, VeriGT is based on FOL, and thus suffers the same expressibility issue as VeriATL (Section 4.5) or any other verifiers that are based on FOL. Nevertheless, we show that by carefully designing the Pacman metamodel, we can use FOL to verify temporal constraints without using formalisms such as DEVS or CTL.

There are also interactive theorem proving approaches for GT verification. Asztalos et al. use category theory to describe graph rewriting systems [5]. This approach is implemented in the VMTS verification framework, but it is not targeted to a specific graph rewriting-based model transformation language. Schätz presents an approach to verify the structural contracts of GT [103]. The transformation rules are given as a textual description based on a relational calculus. The formalisations of model, metamodel and transformation rules are based on declarative relations in a Prolog style, and target the Isabelle/HOL theorem prover. These approaches rely on encoding the execution semantics of the GT language. On top of that, we are able to address a different challenge. That is we also verify that the execution semantics of GT encoded in Boogie correctly represents its corresponding runtime behaviour (i.e. GT implementation), which makes our approach complementary to the existing approaches. Our approach is inspired by the translation validation approach used in compiler verification [97]. An earlier proposal to adapt the translation validation approach in GT verification was also made by Horváth [59].

The most similar work to ours is Poskitt and Plump's work on a Hoare calculus for the correctness of so called graph programs (i.e. graph rules based on double push-out with basic control structures) [100]. We are both following Hoare logic for program verification. One of the main differences is that they reason about the soundness of their verification system against the operational semantics of the GT language, which is more suitable for GT languages that are without a transformation implementation. We chose to reason about the soundness against the runtime behaviour of the GT implementation, which is more suitable for interpreted GT languages. Another main difference is that they manually prove the soundness of their verification system by induction, whereas we chose the translation validation approach to automatically verify the soundness of our verification system. Later on, Poskitt and Plump identify extra properties (e.g. connectedness, existence of paths) that cannot be proved by standard FOL [101]. Therefore, they propose to extend standard FOL with node- and edge-set quantification, and set membership predicates to prove the identified properties. The responsibility of their FOL extension is similar to our OCL library that enables quantification over collections. We have not checked all their identified challenges for standard FOL (e.g. absence of cycles). However, it is in principle feasible by carefully axiomatising inductive declarations [34], which we would like to natively embed into VeriGT in the future.

6.7 Summary

In this chapter, we have described the design of the VeriGT verifier to soundly prove the partial correctness of SimpleGT graph transformations. Its design demonstrates the versatility of our VeriMTLr framework which facilitates verifier construction for both MTr and GT languages. The main contributions of this chapter are a semantics for the SimpleGT language and a semantics for EMFTVM bytecode. In particular, we have demonstrated the difference between the execution semantics of relational and graph transformations, and quantified how the difference would affect their verifier design. We have also illustrated how to develop the semantics of EMFTVM bytecode by extending the semantics of ASM bytecode from the VeriMTLr framework, to extend the translation validation approach to a wider range of model transformation languages (particularly for model transformation languages with explicit memory deallocation). Finally, we have evaluated the VeriGT verifier on the well-known Pacman game, which shows its performance and feasibility.

VeriGT does not support SimpleGT transformations with rule inheritance. However, such an input transformation can be de-sugared into a behaviour-equivalent inheritance-free transformation and then we can apply VeriGT. In addition, VeriGT cannot verify the total correctness of SimpleGT, since we still experiment the details regarding graph transformation termination.

Chapter 7

Conclusion

In summary, this thesis is about the correctness of MTr. We aim at making its formal verification in practice reliable and feasible, by modularly designing reusable and sound verification systems.

The thesis validated the possibility of using an IVL to systematically design modular and reusable verifiers for the given target MTr language. We have systematically designed the VeriATL verifier using the Boogie IVL to verify the correctness of the ATL language. In the process, we have modularly constructed several Boogie libraries, under the hood of our VeriMTLr framework, that can be reused across different verifier designs.

Striving to make verification reliable, we confronted the potential unsoundness of verifiers, and used a translation validation approach to solve this problem. This is not possible without the proposed translational semantics in Boogie to precisely explain the runtime behaviour of the underlying transformation implementation, which is also encapsulated in the VeriMTLr framework, facilitating reuse.

Our curiosity further drove us to extend our work to a GT language, namely SimpleGT. We demonstrated the differences between the execution semantics of relational and graph transformations, and quantified how the differences would affect their verifier designs.

There are lessons that we learned during the journey of our research, and there is more work to be done before mature systems, that are routinely used by modellers and programmers to ensure the correctness of model transformations, can be developed. This chapter is dedicated to demonstrating them.

7.1 Observations from our Research

In this section, we discuss two observations obtained while developing the VeriMTLr framework.

7.1.1 Interoperability between Verifiers

Interoperability between verifiers allows several theorem provers to interact with each other (e.g. share information, chain the proof obligations) so that they can collectively prove theorems more automatically than any one of them could prove in isolation.

Kaufmann and Moore, based on their experience, suggest that what prevents interoperability between verifiers is that the time it takes to interact with another verifier is often dominated by the time it takes to convert a problem into the representation used by the "foreign" verifier [67].

Thus, verifiers that are based on the same IVL provide the foundation to enable their interoperability. We also find that verifiers based on the same memory model further enhance their interoperability, since the elements in the memory model are organised in the same fashion, and conform to the same principles to access and update the elements. Consequently, it enables a natural embedding from one verifier to the other.

We have applied the Dafny and VeriATL verifiers on the ER2REL transformation to demonstrate the possibility of interoperation between the two verifiers. Both verifiers are based on the Boogie IVL.

Dafny. The Dafny language is designed to support the static verification of both procedure-oriented and object-oriented programs [82]. It is imperative, sequential, and it supports generic classes, dynamic allocation, and inductive datatypes, and built-in contract constructs. The contracts include pre/postconditions, frame conditions and termination metrics. To further support contracts, Dafny also offers ghost variables, recursive functions, and mathematical types like sets and sequences.

The Dafny code shown in Fig. 7.1 introduces the *ERSchema* and *RELSchema* classes of Fig. 4.1 (Line 1 - 5). Both classes encapsulate a *name* field, which declares the data that the instance of each class can carry. Next, we declare a *ER2REL* class that encapsulates the transformation between the two introduced classes. To enhance the readability and modularity of the Dafny code, the precondition and postcondition of the transformation are specified in the predicate *pre* and *post* respectively. The specified *read* clauses acquire the read permission for the predicate parameters. Finally, the workhorse of the transformation is represented as a unimplemented method (i.e. no method body) from line 23 - 26, which

```
class ERSchema
{ var name : string; }

class RELSchema
{ var name: string; }

class ER2REL
{
  /* precondition of ER2REL: unique name of ERSchema */
  predicate pre(er: set<ERSchema>)
    reads er;
  {
    ∀ s1, s2: ERSchema •
      s1 in er ∧ s2 in er ∧ s1≠s2 ∧ s1≠null ∧ s2≠null ⟹ s1.name ≠ s2.name }

  /* postcondition of ER2REL: unique name of RELSchema */
  predicate post(rel: set<RELSchema>)
    reads rel;
  {
    ∀ r1, r2: RELSchema •
      r1 in rel ∧ r2 in rel ∧ r1≠r2 ∧ r1≠null ∧ r2 ≠ null ⟹ r1.name ≠ r2.name}

  /* unimplemented model transformation */
  method ER2REL(er: set<ERSchema>) returns (rel: set<RELSchema>)
    requires pre(er);
    ensures post(rel);

}
```

Fig. 7.1 Dafny classes for ER, REL and ER2REL

we intend to build using an ATL transformation (as demonstrated in Fig. 4.2 of Chapter 4).

Next, the Dafny developer can proceed with their verification activities presuming the eventual existence of verified *ER2REL* transformation. Simultaneously, the ATL developer will develop the transformation and use VeriATL to reason about its correctness. The contracts of *ER2REL* transformation used by the VeriATL reuse the compiled Boogie code that corresponds to the Dafny contracts of the *ER2REL* method (shown in Fig. 7.2). Thus, the ATL developer does not need to rewrite the contracts again in OCL, or worry about whether the contracts they produced represent the assumptions made by the Dafny developer. In addition, the interoperability can go both ways, i.e. VeriATL also takes advantage of the static verification capability of Dafny to ensure the input model satisfies the precondition of *ERSchema* before executing the transformation.

However, we anticipate that contract reuse will not be as easy as it seems. The major difficulty comes from exporting theories. To allow proof obligations to be understood by the interoperated verifier, the caller verifier is required to export its theories to the callee verifier. The exporting task could just require the syntactic rewritings to bridge between theories, e.g. the Boogie expression $Is(o,Ty())$ from Dafny is equivalent to $dtype(o) == Ty$ in VeriATL.

Exporting theories becomes more challenging if semantic issues are involved. For ex-

```
// definition axiom for _module.ER2REL.pre (intra-module)
... (∀ s1#0: ref, s2#0: ref •
  Is(s1#0, Tclass._module.ERSchema()) ∧ IsAlloc(s1#0, Tclass._module.ERSchema(), $Heap)
  ∧ Is(s2#0, Tclass._module.ERSchema()) ∧ IsAlloc(s2#0, Tclass._module.ERSchema(), $Heap)
    ⟹ er#0[$Box(s1#0)] ∧ er#0[$Box(s2#0)]
     ∧ s1#0 ≠ s2#0 ∧ s1#0 ≠ null ∧ s2#0 ≠ null ⟹
      ¬Seq#Equal(read($Heap, s1#0, _module.ERSchema.name),
                 read($Heap, s2#0, _module.ERSchema.name))) ...

// definition axiom for _module.ER2REL.post (intra-module)
... (∀ r1#0: ref, r2#0: ref •
  Is(r1#0, Tclass._module.RELSchema()) ∧ IsAlloc(r1#0, Tclass._module.RELSchema(), $Heap)
  ∧ Is(r2#0, Tclass._module.RELSchema()) ∧ IsAlloc(r2#0, Tclass._module.RELSchema(), $Heap)
    ⟹ rel#0[$Box(r1#0)] ∧ rel#0[$Box(r2#0)]
     ∧ r1#0 ≠ r2#0 ∧ r1#0 ≠ null ∧ r2#0 ≠ null ⟹
      ¬Seq#Equal(read($Heap, r1#0, _module.RELSchema.name),
                 read($Heap, r2#0, _module.RELSchema.name))) ...
```

Fig. 7.2 Boogie code for the transformation contracts, as generated by Dafny

```
axiom (∀ s1#0: ref •
  Is(s1#0, Tclass._module.ERSchema())
  ∧ IsAlloc(s1#0, Tclass._module.ERSchema(), $Heap)
  ∧ s1#0 ≠ null ⟺
    dtype(s1#0) = ER$ERSchema
    ∧ read(srcHeap, s1#0, alloc)
    ∧ s1#0≠null);

axiom (∀<α>, s1#0: ref, f: Field α •
  Is(s1#0, Tclass._module.ERSchema())
  ∧ IsAlloc(s1#0, Tclass._module.ERSchema(), $Heap)
  ∧ s1#0 ≠ null ⟹
       read($Heap, s1#0, f) = read(srcHeap, s1#0, f) );
```

Fig. 7.3 Auxiliary Boogie axioms to bridge memory models between Dafny and VeriATL

ample, the *ER2REL* class in Fig. 7.1 uses two arrays *er* and *rel* to distinguish the source and target of the transformation; however, they are allocated on the same heap as shown in Fig. 7.2. In contrast, VeriATL uses separate heaps to organise source and target models. Thus, the differences between the theories of the caller and callee verifiers must be handled carefully. We believe this can be done by providing auxiliary axioms. For example, in Fig. 7.3, the first axiom states that all the allocated instances of ERSchema on the *Heap* of Dafny are allocated with equivalent types on the *srcHeap* of VeriATL. The second axiom states that the fields of all the allocated instances of ERSchema on the *Heap* of Dafny and their correspondents in VeriATL agree on their values. One caveat here is that the second axiom can be dangerous in the case of two verifiers with different type systems, which could result in the need for more axioms to specify the type discrepancy.

In conclusion, we suggest that it will be a long, but promising journey towards witnessing a full verification ecosystem. The verifiers in this verification ecosystem will adapt the same IVL using the same memory model, and aim to be orchestrated to collectively prove the correctness of programs more automatically.

7.1.2 Verifiability of Model Transformation Languages

In this research, one of our main goals is to design sound verifiers that are based on FOL. We refer to the “verifiability” of a model transformation language as the degree of difficulty in designing a sound verifier for the given model transformation language. Thus, low verifiability means it is difficult to design a sound verifier for the given model transformation language, whereas high verifiability means it is easy to design a sound verifier for the given model transformation language.

Through our development experience, both implicit semantics and counter-intuitive semantics are two factors that compromise verifiability.

By “implicit semantics”, we mean that there are extra semantics hidden in the language design which cannot be revealed easily (e.g. by examining the language specification). We pick the *SET* instructions from the ASM language to demonstrate this phenomenon. It is tempting to design the semantics of the *SET* instruction as “set the structural feature of the given element to the given value”. What is lurking there is that the *SET* instruction behaves differently when the structural feature being set is an association whose multiplicity has an upper bound greater than one. Missing the hidden semantics of the *SET* instruction has direct consequences on the soundness of verifier design (Section 5.2). However, it is not easy to identify. We found the hidden semantics of the *SET* instruction by carefully examining the Java implementation of the ASM virtual machine.

Ideally the solution would be to always have a detailed language specification at hand while developing the verifier for the model transformation language. However, in certain circumstances, the language design could intrinsically make the verifier design difficult. In that case, we prefer evolving the model transformation language for higher verifiability. For example, the *SET* instruction can be decomposed into two instructions to make both aspects of its original semantics explicit (i.e. one instruction for an attribute or association whose multiplicity has an upper bound not greater than one, and another instruction for an association whose multiplicity has an upper bound greater than one).

By “counter-intuitive semantics”, we mean that the language semantics is counter-intuitive to the common understanding of the majority of programmers. Take the consecutive bindings in ATL for example. A Java programmer might intuitively assume that the second binding overwrites the first binding, just like two consecutive assignments. However, when the bound element is an association whose multiplicity has an upper bound greater than one, ATL composes the second binding with the first binding. This can be confusing when developing a verifier for ATL.

We believe that the counter-intuitive semantics is like an anti-design-pattern in software engineering, whose characteristics can be quantified. Then, they can be removed by refactoring. Not only this will make the transformation easier to maintain, but it will also increase the verifiability of the model transformation language, since the counter-intuitive semantics is diminished after refactoring. For example, Wimmer et al. propose a list of refactorings to increase the maintainability of ATL transformations [121]. They demonstrate the symptoms of 24 anti-patterns in ATL transformations, and state the refactoring steps to remove them. How to adapt their refactoring approach to remove counter-intuitive semantics in a model transformation language, and even prove the refactoring preserves the behaviours of the original transformation, would be interesting for further research.

Thus, our hypothesis is that avoiding implicit semantics and counter-intuitive semantics is inevitable for a model transformation language design with a goal of high verifiability.

In our experience, whether model transformation languages are relational or functional intrinsically increases their verifiability. Relational or functional model transformation languages prohibit imperative language constructs that yield side effects. For example, when we designed VeriATL, we considered ATL matched rules in particular, which are always propagated on an empty target model that is disjoint from the source model. Thus, we were able to use two separate *heaps* to organise the source and target elements. This ensures, for example, a modification made on the target *heap* will have no side effects on the state of the source *heap*. Therefore, it yields a simpler encoding that would contribute to automating the translation validation approach.

Languages that are domain specific also increase their verifiability, because they have unified deterministic goals to achieve. Thus, it is easier to abstract their semantics than abstracting the semantics of an imperative program that achieves an arbitrary goal. This feature greatly reduces the complexity of adapting the translation validation approach and enables its automation.

Therefore, we anticipate that relational and functional model transformation languages, or domain specific model transformation languages will benefit most from adapting our VeriMTLr framework to design a verifier which can verify their correctness.

7.2 Future Work

In addition to the future work that we identified in the previous chapters (i.e. evaluating the verification performance on different encodings of our Boogie libraries for improving their completeness, covering more ATL features to build upon the current VeriATL verifier, and implementing VeriGT with the capability of termination proof and the inductive declarations), we also like to focus on four directions to extend our current efforts.

7.2.1 Modular and Reusable Verifier Design for other Model Transformation Languages

In this research, we use the VeriMTLr framework to design a verifier for the ATL MTr language and the SimpleGT GT language. Other MTr languages, in principle, can exploit the facilities provided by the framework to design their verifiers which support the verification of partial correctness, termination analysis and other logical reasoning tasks.

On top of this adaptation, our future work will focus on enabling another form of reusable verifier design by means of higher-order transformation (i.e. applying model transformations on model transformations [15]). Higher-order transformation involves developing a transformation to translate the model transformation that is written in the language T_1 into a model transformation that is written in a language T_2, where the verifier of T_2 already exists (developed by our framework, or developed by a third party). For example, a MTr can be reduced to a GT in the SimpleGT language by means of a triple graph grammar [104]. This allows the MTr to be verified without developing a new verifier for it. However, this is not always possible due to the fundamental semantic differences of involved model transformation languages, which is why we cannot express a SimpleGT transformation in terms of an ATL transformation to reuse VeriATL. In addition, developing high-order transformations is also a non-trivial task, since the domain-specific properties of the model transformation languages need to be studied carefully [54]. Therefore, we anticipate that our future work will include the development of a new language to enable systematic and correct high-order transformations.

7.2.2 Verifying Model Transformations for Programming Language Transformation

The case studies we presented in this research are data-centric model transformations. In a broader sense, programs are models as well. Therefore, a program transformation can be viewed as a model transformation. The reason that they are usually treated separately is because program transformation itself has a mature and dedicated community for this specific task [1].

In [29], we show that the task of transforming Event-B programs to Boogie programs can be written as an ATL model transformation. After we enhance the capability of VeriATL to handle lazy matched rules, we would like to apply it to check the correctness of Event-B to Boogie transformations. We anticipate the main challenge will come from formulating semantic correctness contracts. The main semantic correctness contract we would like to prove is that the transformation always generates the target model that preserves the possible behaviour (e.g. that the program terminates, or a trace of memory updating) of the source model. For this verification task, there are some questions awaiting us:

- Should the transformation allow additional possible behaviour in the target model?
- Would there be a semantic framework to ease expressing the possible behaviour? e.g. an abstract state machine [19].

We expect to prove by construction that the current version of OCL that we support is capable of developing such a semantic framework.

7.2.3 Axiomatic System Consistency Verification

Currently, to detect inconsistency in our Boogie libraries, we predefine oracles, and use the smoke option in Boogie [91]. However, there is no guarantee that the inconsistent axioms will be revealed.

Given a theory with a list of axioms, one can use a theorem prover to make a realisation of this theory. This shows the relative consistency of such an axiomatised theory by making a connection to an existing library of the proof assistant/theorem prover. This technique is known as the theory interpretation [48].

There already exists a variety of work on applying theory interpretation for non-trivial existing theories. For example, realisations of Why3 theories are supported for the proof assistant Coq [13] and PVS [108]. Darvas and Müller investigate the consistency of a model class in JML [43], by exporting it to the proof assistant Isabelle [92]. Compared to the realisations of a Why3 theory, Darvas and Müller explicitly design additional proof obligations to cope with the properties brought with the model class (e.g. to ensure the type of a model class and its counterpart in the target theorem prover are compatible). Thus, existing work would provide guidance for us to include this useful feature in our Boogie libraries.

7.2.4 Generating Counter-examples on Verification Failure

On verification failure, the trace information from the Boogie verifier, indicating where the incorrectness (within the input Boogie program) was detected, will be output. At the moment, we do not perform any re-construction of such trace information, which is a disadvantage, since we cannot report useful feedback to a non-Boogie expert on verification failure.

Our main concern is that the gross profit of re-constructing trace information is low. The trace information indicates a list of execution points which can be safely reached without violating the enforced constraints on those points, before finally reaching where the incorrectness was detected. In our experience, such information is very useful when debugging the Boogie program by inserting additional assertions to understand it better, getting closer to the exact issue that causes the verification failure. However, from the perspective of a model transformation developer, it is unlikely that they could infer much useful information from this trace information. What would be useful is to find a concrete counter-example to reproduce the verification failure.

Model finding, whose goal is to find models that satisfy the given constraints, is particularly suitable for generating counter-examples. We anticipate that the main challenges here could be to encode the domain specific properties of models (e.g. properties of EMF), and to express a set of precise constraints that need to be satisfied by the models. For the first challenge, Wu et al. present a model finding approach, based on the attributed type graph, specifically tailored for MDE [123]. How to incorporate their approach within our framework will be among our highest priorities. For the second challenge, we conjecture that the trace information will provide the required constraints for model finding. However, how the trace information is represented as the required constraints, and whether the trace information is all we need for finding the proposed counter-example demands further investigation.

References

[1] Ab.Rahim, L. and Whittle, J. (2015). A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028.

[2] Amrani, M., Lucio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y., and Cordy, J. R. (2012). A tridimensional approach for studying the formal verification of model transformations. In *5th International Conference on Software Testing, Verification and Validation*, pages 921–928, Washington, DC, USA. IEEE Computer Society.

[3] Anastasakis, K., Bordbar, B., and Küster., J. M. (2007). Analysis of model transformations via Alloy. In *4th Workshop on Model-Driven Engineering, Verification and Validation*, pages 47–56. Nashville, TN, USA.

[4] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced concepts and tools for in-place EMF model transformations. In *13th International Conference on Model Driven Engineering Languages and Systems*, pages 121–135, Oslo, Norway. Springer.

[5] Asztalos, M., Lengyel, L., and Levendovszky, T. (2013). Formal specification and analysis of functional properties of graph rewriting-based model transformation. *Software Testing, Verification and Reliability*, 23(5):405–435.

[6] ATLAS Group (2005). Specification of the ATL virtual machine. Technical report, Lina & INRIA Nantes.

[7] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A modular reusable verifier for object-oriented programs. In *4th International Conference on Formal Methods for Components and Objects*, pages 364–387, Amsterdam, Netherlands. Springer.

[8] Barnett, M., Leino, K. R. M., and Schulte, W. (2005). The Spec# programming system: An overview. In *1st International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Marseille, France. Springer.

[9] Barroca, B., Lúcio, L., Amaral, V., Félix, R., and Sousa, V. (2011). DSLTrans: A turing incomplete transformation language. In *3rd International Conference on Software Language Engineering*, pages 296–305, Eindhoven, Netherlands. Springer.

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

[11] Bergmann, G. (2014). Translating OCL to graph patterns. In *17th International Conference on Model-Driven Engineering Languages and Systems*, pages 670–686, Valencia, Spain. Springer.

[12] Berry, G. (2008). Synchronous design and verification of critical embedded systems using SCADE and Esterel. In *12th International Workshop on Formal Methods for Industrial Critical Systems*, pages 2–2. Springer, Berlin, Germany.

[13] Bertot, Y. and Castran, P. (2010). *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer, 1st edition.

[14] Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188.

[15] Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., and Lindow, A. (2006). Model transformations? transformation models! In *9th International Conference on Model Driven Engineering Languages and Systems*, pages 440–453, Genova, Italy. Springer.

[16] Bill, R., Gabmeyer, S., Kaufmann, P., and Seidl, M. (2014). Model checking of CTL-extended OCL specifications. In *7th International Conference on Software Language Engineering*, pages 221–240, Västerås, Sweden. Springer.

[17] Boehm, B. W. (1984). Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88.

[18] Böhme, S. and Moskal, M. (2011). Heaps and data structures: A challenge for automated provers. In *23rd International Conference on Automated Deduction*, pages 177–191, Wroclaw, Poland. Springer.

[19] Borger, E. and Stark, R. F. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer.

[20] Bornat, R. (2000). Proving pointer programs in Hoare logic. In *International Conference on Mathematics of Program Construction*, pages 102–126, Ponte de Lima, Portugal. Springer.

[21] Büttner, F., Egea, M., Cabot, J., and Gogolla, M. (2012a). On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *15th International Conference on Model Driven Engineering Languages and Systems*, pages 198–213, Innsbruck, Austria. Springer.

[22] Büttner, F., Egea, M., Cabot, J., and Gogolla, M. (2012b). Verification of ATL transformations using transformation models and model finders. In *14th International Conference on Formal Engineering Methods*, pages 198–213, Kyoto, Japan. Springer.

[23] Büttner, F., Egea, M., Guerra, E., and de Lara, J. (2013). Checking model transformation refinement. In *6th International Conference on Model Transformation*, pages 158–173, Budapest, Hungary. Springer.

[24] Cabot, J., Clarisó, R., Guerra, E., and de Lara, J. (2010). Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302.

[25] Calegari, D., Luna, C., Szasz, N., and Tasistro, Á. (2011). A type-theoretic framework for certified model transformations. In *13th Brazilian Symposium on Formal Methods*, pages 112–127, Natal, Brazil. Springer.

[26] Calegari, D. and Szasz, N. (2013). Verification of model transformations: A survey of the state-of-the-art. *Electronic Notes in Theoretical Computer Science*, 292(0):5–25.

[27] CASC-J7 (2014). CADE ATP system competition. http://www.cs.miami.edu/~tptp/CASC/.

[28] Chan, K. (2006). Formal proofs for QoS-oriented transformations. In *10th International Conference Workshops on Enterprise Distributed Object Computing*, pages 41–41, Hong Kong, China. IEEE.

[29] Cheng, Z., Monahan, R., and Power, J. F. (2012). Machine-checked refinement proof obligations for Rodin in the Boogie2 language. Technical report, Maynooth University, Ireland.

[30] Cheng, Z., Monahan, R., and Power, J. F. (2013). Online repository for VeriATL system. https://github.com/VeriATL/VeriATL.

[31] Cheng, Z., Monahan, R., and Power, J. F. (2015a). Online repository for VeriGT system. https://github.com/VeriATL/VeriGT.

[32] Cheng, Z., Monahan, R., and Power, J. F. (2015b). A sound execution semantics for ATL via translation validation. In *8th International Conference on Model Transformation*, pages 133–148, L'Aquila, Italy. Springer.

[33] Cheng, Z., Monahan, R., and Power, J. F. (2015c). Verifying SimpleGT transformations using an intermediate verification language. In *4th International Workshop on the Verification Of Model Transformation*, page To appear, L'Aquila, Italy. CEUR.

[34] Clochard, M., Filliâtre, J. C., Marché, C., and Paskevich, A. (2014). Formalizing semantics with an automatic program verifier. In *6th International Conference on Verified Software: Theories, Tools and Experiments*, pages 37–51, Vienna, Austria. Springer.

[35] Combemale, B., Crégut, X., Garoche, P., and Thirioux, X. (2009). Essay on semantics definition in MDE - an instrumented approach for model verification. *Journal of Software*, 4(9):943–958.

[36] Correnson, L., Cuoq, P., Puccetti, A., and Signoles, J. (2010). Frama-C user manual. *CEA LIST*, page 111.

[37] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM.

[38] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

[39] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., and Schulte, W. (2009). VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering*, pages 429–430, Vancouver, British Columbia. IEEE.

[40] Darvas, Á. and Leino, K. R. M. (2007). Practical reasoning about invocations and implementations of pure methods. In *10th International Conference on Fundamental Approaches to Software Engineering*, pages 336–351, Braga, Portugal. Springer.

[41] Darvas, Á. and Müller, P. (2006). Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85.

[42] Darvas, Á. and Müller, P. (2008). Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499.

[43] Darvas, Á. and Müller, P. (2010). Proving consistency and completeness of model classes using theory interpretation. In *13th International Conference on Fundamental Approaches to Software Engineering*, pages 218–232, Paphos, Cyprus. Springer.

[44] de Moura, L. and Bjørner, N. (2008). 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. Springer.

[45] Delahaye, D., Dubois, C., Marché, C., and Mentré, D. (2014). The BWare project: Building a proof platform for the automated verification of B proof obligations. In *4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 290–293, Toulouse, France. Springer.

[46] Detlefs, D., Nelson, G., and Saxe, J. B. (2005). Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473.

[47] Egea, M. and Büttner, F. (2014). Verification of authorization policies modified by delegation. In *Engineering Secure Future Internet Services and Systems - Current Research*, pages 287–314.

[48] Farmer, W. M. (1994). Theory interpretation in simple type theory. In *1st International Workshop on Higher-Order Algebra, Logic, and Term Rewriting*, pages 96–123, Amsterdam, The Netherlands. Springer.

[49] Fernández, M. and Terrell, J. (2013). Assembling the proofs of ordered model transformations. In *10th International Workshop on Formal Engineering approaches to Software Components and Architectures*, pages 63–77, Rome, Italy. EPTCS.

[50] Filliâtre, J. C. (2003). Why: A multi-language multi-prover verification tool. Technical report, LRI, Université Paris Sud.

[51] Filliâtre, J. C. (2013). One logic to use them all. In *24th International Conference on Automated Deduction*, pages 1–20, Lake Placid, NY, USA. Springer.

[52] Filliâtre, J. C. and Paskevich, A. (2013). Why3 — where programs meet provers. In *22nd European Symposium on Programming*, pages 125–128, Rome, Italy. Springer.

[53] Ge, Y., Barrett, C., and Tinelli, C. (2009). Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122.

[54] Greenyer, J. and Kindler, E. (2010). Comparing relational model transformation technologies: implementing Query/View/Transformation with triple graph grammars. *Software & Systems Modeling*, 9(1):21–46.

[55] Gregoire, B. and Melquiond, G. (2013). Why3 (ver. 0.80) bug with Yices. Why3-club mailing list Vol 27, Issue 6.

[56] Guerra, E. and de Lara, J. (2014). Colouring: execution, debug and analysis of QVT-relations transformations through coloured Petri nets. *Software & Systems Modeling*, 13(4):1447–1472.

[57] Hoang, D., Moy, Y., Wallenburg, A., and Chapman, R. (2014). SPARK 2014 and GNATprove. *International Journal on Software Tools for Technology Transfer*, (Preprint).

[58] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

[59] Horváth, Á. (2008). Towards a two layered verification approach for compiled graph transformation. In *4th International Conference on Graph Transformations*, pages 499–501, Leicester, United Kingdom. Springer.

[60] Howard, W. A. (1980). The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press.

[61] Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press.

[62] Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290.

[63] Jackson, E. K., Levendovszky, T., and Balasubramanian, D. (2011). Reasoning about metamodeling with formal specifications and automatic proofs. In *14th International Conference on Model driven engineering languages and systems*, pages 653–667, Wellington, New Zealand. Springer.

[64] Jackson, P., Schanda, F., and Wallenburg, A. (2013). Auditing user-provided axioms in software verification conditions. In *18th International Workshop on Formal Methods for Industrial Critical Systems*, pages 154–168, Madrid, Spain. Springer.

[65] Jézéquel, J.-M., Barais, O., and Fleurey, F. (2011). Model driven language engineering with Kermeta. In *3rd International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 201–221. Springer, Braga, Portugal.

[66] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39.

[67] Kaufmann, M. and Moore, J. S. (2004). Some key research problems in automated theorem proving for hardware and software verification. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales. Serie A. Matemâticas*, 98(1):181–195.

[68] Klatt, B. (2007). Xpand: A closer look at the model2text transformation language. http://bar54.de/benjamin.klatt-Xpand.pdf.

[69] Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman, Boston, MA, USA.

[70] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2008). The Epsilon transformation language. In *1st International Conference on Model Transformations*, pages 46–60. Springer, Zürich, Switzerland.

[71] Kuhlmann, M., Hamann, L., and Gogolla, M. (2011). Extensive validation of OCL models by integrating SAT solving into USE. In *49th International Conference on Objects, Models, Components, Patterns*, pages 290–306, Zürich, Switzerland. Springer.

[72] Lano, K. (2006). Using B to verify UML transformations. In *3rd Workshop on Model-Driven Engineering, Verification and Validation*, pages 46–61. Genova, Italy.

[73] Lano, K. (2015). UML-RSDS toolset and manual. http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf.

[74] Lano, K., Clark, T., and Kolahdouz-Rahimi, S. (2014a). A framework for model transformation verification. *Formal Aspects of Computing*, 27(1):193–235.

[75] Lano, K. and Kolahdouz-Rahimi, S. (2010). Specification and verification of model transformations using UML-RSDS. In *8th International Conference on Integrated Formal Methods*, pages 199–214, Nancy, France. Springer.

[76] Lano, K. and Kolahdouz-Rahimi, S. (2013). Constraint-based specification of model transformations. *Journal of Systems and Software*, 86(2):412–436.

[77] Lano, K., Kolahdouz-Rahimi, S., and Clark, T. (2012). Comparing verification techniques for model transformations. In *9th Workshop on Model-Driven Engineering, Verification and Validation*, pages 23–28, Innsbruck, Austria. ACM.

[78] Lano, K., Kolahdouz-Rahimi, S., Poernomo, I., Terrell, J., and Zschaler, S. (2014b). Correct-by-construction synthesis of model transformations using transformation patterns. *Software & Systems Modeling*, 13(2):873–907.

[79] Ledang, H. and Dubois, H. (2010). Proving model transformations. In *4th International Symposium on Theoretical Aspects of Software Engineering*, pages 35–44, Taipei, Taiwan. IEEE.

[80] Lehner, H. and Müller, P. (2007). Formal translation of bytecode into BoogiePL. In *2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, pages 35–50, Budapest, Hungary. Elsevier.

[81] Leino, K. R. M. (2008). *This is Boogie 2. http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf*. Microsoft Research, Redmond, USA.

[82] Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Dakar, Senegal. Springer.

[83] Leino, K. R. M. and Middelkoop, R. (2009). Proving consistency of pure methods and model fields. In *12th International Conference on Fundamental Approaches to Software Engineering*, pages 231–245, York, UK. Springer.

[84] Leino, K. R. M. and Monahan, R. (2009). Reasoning about comprehensions with first-order SMT solvers. In *24th Annual ACM Symposium on Applied Computing*, pages 615–622, Hawaii, USA. ACM.

[85] Leino, K. R. M. and Rümmer, P. (2010). A polymorphic intermediate verification language: Design and logical encoding. In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327, Paphos, Cyprus. Springer.

[86] Leroy, X. (2006). Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Notices*, 41(1):42–54.

[87] Lúcio, L., Barroca, B., and Amaral, V. (2010). A technique for automatic validation of model transformations. In *13th International Conference on Model driven engineering languages and systems*, pages 136–150, Oslo, Norway. Springer.

[88] Lúcio, L. and Vangheluwe, H. (2013). Model transformations to verify model transformations. In *2nd Workshop on Verification of Model Transformations*. Budapest, Hungary.

[89] Manna, Z. and McCarthy, J. (1969). Properties of programs and partial function logic. *Machine Intelligence*, 5:27–38.

[90] Marché, C., Paulin-Mohring, C., and Urbain, X. (2004). The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1-2):89–106.

[91] Moskal, M. (2009). Programming with triggers. In *7th International Workshop on Satisfiability Modulo Theories*, pages 20–29, Montreal, Canada. ACM.

[92] Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer.

[93] Object Management Group (2000). OMG unified modeling language specification (ver. 1.3). http://doc.omg.org/formal/2000-03-01.pdf.

[94] Object Management Group (2011). Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification (ver. 1.1). http://www.omg.org/spec/QVT/1.1.

[95] Plump, D. (1998). Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209.

[96] Plump, D. (2005). Confluence of graph transformation revisited. In *Processes, Terms and Cycles*. Springer.

[97] Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK. Springer.

[98] Poernomo, I. (2008). Proofs-as-model-transformations. In *1st International Conference on Model Transformation*, pages 214–228, Zürich, Switzerland. Springer.

[99] Poernomo, I. and Terrell, J. (2010). Correct-by-construction model transformations from partially ordered specifications in Coq. In *12th International Conference on Formal Engineering Methods*, pages 56–73, Shanghai, China. Springer.

[100] Poskitt, C. and Plump, D. (2010). A Hoare calculus for graph programs. In *5th International Conference on Graph Transformations*, pages 139–154, Enschede, The Netherlands. Springer.

[101] Poskitt, C. and Plump, D. (2014). Verifying monadic second-order properties of graph programs. In *7th International Conference Graph Transformation*, pages 33–48, York, UK. Springer.

[102] Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual*. Pearson Education, 2nd edition.

[103] Schätz, B. (2010). Verification of model transformations. In *9th International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 130–142, Paphos, Cyprus. EASST.

[104] Schürr, A. (1995). Specification of graph translators with triple graph grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, Herrsching, Germany. Springer.

[105] Selim, G., Büttner, F., Cordy, J., Dingel, J., and Wang, S. (2013). Automated verification of model transformations in the automotive industry. In *16th International Conference on Model-Driven Engineering Languages and Systems*, pages 690–706, Miami, FL, USA. Springer.

[106] Selim, G., Lúcio, L., Cordy, J., Dingel, J., and Oakes, B. (2014). Specification and verification of graph-based model transformation properties. In *7th International Conference on Graph Transformation*, pages 113–129, York, UK. Springer.

[107] Selim, G., Wang, S., Cordy, J., and Dingel, J. (2012). Model transformations for migrating legacy models: An industrial case study. In *8th European Conference on Modelling Foundations and Applications*, pages 90–101, Lyngby, Denmark. Springer.

[108] Shankar, N., Owre, S., Rushby, J. M., and Stringer-Calvert, D. W. J. (2001). PVS prover guide. http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf.

[109] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse modeling framework*. Pearson Education, 2nd edition.

[110] Stenzel, K. (2004). A formally verified calculus for full Java card. In *10th International Conference on Algebraic Methodology and Software Technology*, pages 491–505, Stirling, Scotland, UK. Springer.

[111] Stevens, P. (2013). A simple game-theoretic approach to checkonly QVT relations. *Software and System Modeling*, 12(1):175–199.

[112] Syriani, E. and Vangheluwe, H. (2013). A modular timed graph transformation language for simulation-based design. *Software & Systems Modeling*, 12(2):387–414.

[113] Troya, J. and Vallecillo, A. (2011). A rewriting logic semantics for ATL. *Journal of Object Technology*, 10(5):1–29.

[114] Tschannen, J., Furia, C. A., Nordio, M., and Meyer, B. (2011). Verifying Eiffel programs with Boogie. *Computing Research Repository*, abs/1106.4700.

[115] Varró, G., Varró, D., and Friedl, K. (2006). Adaptive graph pattern matching for model transformations using model-sensitive search plans. In *1st International Workshop on Graph and Model Transformations*, pages 191–205, Brighton, United Kingdom. Elsevier.

[116] Vaziri, M. and Jackson, D. (2000). Some shortcomings of OCL, the object constraint language of UML. In *34th International Conference on Technology of Object-Oriented Languages and Systems*, page 555, Los Alamitos, CA, USA. IEEE Computer Society.

[117] Wagelaar, D. (2014). Using ATL/EMFTVM for import/export of medical data. In *2nd Software Development Automation Conference*, Amsterdam, Netherlands.

[118] Wagelaar, D., Tisi, M., Cabot, J., and Jouault, F. (2011). Towards a general composition semantics for rule-based model transformation. In *14th International Conference on Model Driven Engineering Languages and Systems*, pages 623–637, Wellington, New Zealand. Springer.

[119] Webster, J. and Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. *Management Information Systems Quarterly*, 26(2):13–23.

[120] Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., and Schwinger, W. (2009). Right or wrong? – verification of model transformations using colored Petri nets. In *9th OOPSLA Workshop on Domain-Specific Modeling*, pages 101–106, Orlando, USA. Helsinki School of Economics.

[121] Wimmer, M., Martínez, S., Jouault, F., and Cabot, J. (2012). A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 11(2):1–40.

[122] Wu, H. (2013). *Automated metamodel instance generation satisfying quantitative constraints*. PhD thesis, Maynooth University.

[123] Wu, H., Monahan, R., and Power, J. (2013). Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *7th International Symposium on Theoretical Aspects of Software Engineering*, pages 175–182, Birmingham, UK. IEEE.

Appendix A

Detailed Boogie Encoding of the Semantics of OCL

The detailed Boogie encoding of the semantics of OCL primitives and collections, that corresponds to Table 3.1 and Table 3.2 in Chapter 3, are listed in Table A.1 and Table A.2 respectively. Their first column lists the three primitive data types. The second column details the supported OCL operations on each primitive data type, whose corresponding Boogie encoding is listed in the third column. The Boogie encoding, that is built on top of the existing Boogie libraries or is newly introduced, is listed in **bold** font.

Data Type	**OCL Operation**	**Boogie Encoding**
OCLBool	and, or, implies, not	&&, ||||, ==>
OCLInteger	<, >, >=, <=, =, <>	<, >, >=, <=, ==, !=
	*, +, -, div(), mod()	*, +, -, div, mod
	abs()	**Integerabs()**
OCLString	s.size()	SeqLength(s: String)
	s1.concat(s2:OCLString)	SeqAppend(s1:String, s2:String)
	s.substring (lower:OCLInteger, upper:OCLInteger)	**StringSubstring (s:String, lower:int, upper:int)**
	s.toUpper(), s.toLower()	**StringToUpper(s:String), StringToLower(s:String)**
	s1.startsWith(s2:OCLString), s1.endsWith(s2:OCLString)	**StringStartsWith(s1:String, s2:String), StringEndsWith(s1:String, s2:String)**

Table A.1 Detailed Boogie encoding of OCL primitives

Data Type	OCL Operation	Boogie Encoding
OCLSet	s1.union(s2:OCLSet), s1.intersection(s2:OCLSet), s1-s2	SetUnion(s1:Set, s2:Set), SetIntersection(s1:Set, s2:Set), SetDifference(s1:Set, s2:Set)
	s.including(o:OCLAny), s.excluding(o:OCLAny)	SetUnionOne(s:Set, o:T), **Set#DifferenceOne(s:Set, o:T)**
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	**Set#Includes(s:Set, o:T),** **Set#Excludes(s:Set, o:T)**
	s.isEmpty(), s.notEmpty()	**Set#IsEmpty(s:Set),** **Set#NotEmpty(s:Set)**
OCLBag	s.including(o:OCLAny), s.excluding(o:OCLAny)	MultiSet#UnionOne(s:MultiSet, o:T), **MultiSet#DifferenceOne(s:MultiSet, o:T)**
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	**MultiSet#Includes(s:MultiSet, o:T),** **MultiSet#Excludes(s:MultiSet, o:T)**
	s.isEmpty(), s.notEmpty()	**MultiSet#IsEmpty(s:MultiSet),** **MultiSet#NotEmpty(s:MultiSet)**

Table A.2 Detailed Boogie encoding of OCL collections

Data Type	OCL Operation	Boogie Encoding
OCLOrderedSet	s.append(o:OCLAny), s.prepend(o:OCLAny)	**OrderedSet#Append(s:Seq, o:T),** **OrderedSet#Prepend(s:Seq, o:T)**
	s.insertAt(n:OCLInteger, o:OCLAny)	**OrderedSet#InsertAt(s:Seq,n:int,o:T)**
	s.subOrderedSet(lower:OCLInteger, upper:OCLInteger)	**Seq#Subsequence(s:Seq,lower:int,upper:int)**
	s.at(n:OCLInteger)	**Seq#Index(s:Seq, n:int)**
	s.first(), s.last()	**Seq#First(s:Seq),** **Seq#Last(s:Seq)**
	s.size()	**Seq#Length(s:Seq)**
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	**Seq#Contains(s:Seq, o:T),** **Seq#NotContains(s:Seq, o:T)**
	s.isEmpty(), s.notEmpty()	**Seq#IsEmpty(s:Set),** **Seq#NotEmpty(s:Set)**
OCLSequence	s1.union(s2:OCLSequence)	Seq#Append(s1:Seq,s2:Seq)
	s.append(o:OCLAny), s.prepend(o:OCLAny)	Seq#Build(s:Seq, o:T), **Seq#Prepend(s:Seq, o:T)**
	s.insertAt(n:OCLInteger, o:OCLAny)	**Seq#InsertAt(s:Seq,n:int,o:T)**
	s.subSequence(lower:OCLInteger, upper:OCLInteger)	**Seq#Subsequence(s:Seq,lower:int,upper:int)**
	s.at(n : OCLInteger)	Seq#Index(s:Seq, n:int)
	s.first(), s.last()	**Seq#First(s:Seq),** **Seq#Last(s:Seq)**
	s.size()	Seq#Length(s:Seq)
	s.includes(o:OCLAny), s.excludes(o:OCLAny)	Seq#Contains(s:Seq, o:T), **Seq#NotContains(s:Seq, o:T)**
	s.isEmpty(), s.notEmpty()	**Seq#IsEmpty(s:Set),** **Seq#NotEmpty(s:Set)**

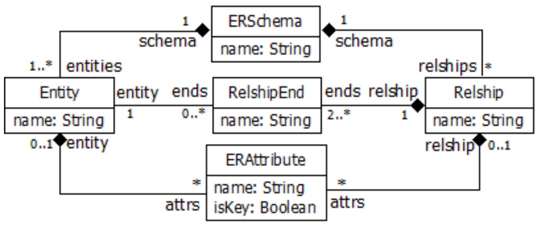
Table A.2 Detailed Boogie encoding of OCL collections (cont.)

Appendix B

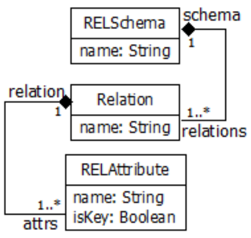
ER2REL and HSM2FSM Transformations in ATL

B.1 ER2REL Transformation in ATL

B.1.1 Entity-Relationship and Relational Metamodels



(a) ER



(b) REL

B.1.2 ATL Transformation for ER2REL

```
module ER2REL; create OUT : REL from IN : ER;

rule S2S {
  from s: ER!ERSchema
  to t: REL!RELSchema (name<-s.name, relations<-s.entities, relations<-s.relships} )}

rule E2R {
  from s: ER!Entity to t: REL!Relation ( name<-s.name) }

rule R2R {
  from s: ER!Relship to t: REL!Relation ( name<-s.name) }

rule EA2A {
  from att: ER!ERAttribute, ent: ER!Entity (att.entity=ent)
  to t: REL!RELAttribute ( name<-att.name, isKey<-att.isKey, relation<-ent ) }

rule RA2A {
  from att: ER!ERAttribute, rs: ER!Relship ( att.relship=rs )
  to t: REL!RELAttribute ( name<-att.name, isKey<-att.isKey, relation<-rs ) }

rule RA2AK {
  from att: ER!ERAttribute, rse: ER!RelshipEnd
      ( att.entity=rse.entity and att.isKey=true )
  to t: REL!RELAttribute ( name<-att.name, isKey<-att.isKey, relation<-rse.relship )}
```

B.1.3 OCL Contracts for ER2REL

```
-- PRECONDITION OF ER

context ER!ERSchema inv unique_er_schema_names: -- unique name of ERSchemas
  ER!ERSchema.allInstances()->forAll(s1,s2| s1<>s2 implies s1.name<>s2.name)

context ER!ERSchema inv unique_er_relship_names: -- relship names are unique in ERSchema
  ER!ERSchema.allInstances()->forAll(s | s.relships->forAll(r1,r2 | r1<>r2 implies r1.name<>r2.name))

context ER!ERSchema inv unique_er_entity_names: -- entity names are unique in ERSchema
  ER!ERSchema.allInstances()->forAll(s | s.entities->forAll(e1,e2 | e1<>e2 implies e1.name<>e2.name))

context ER!ERSchema inv disjoint_er_entity_relship_names: -- disjoint entity and relship names
  ER!ERSchema.allInstances()->forAll(s | s.relships->forAll(r |
    s.entities->forAll(e | r<>e implies r.name<>e.name)))

context ER!Entity inv unique_er_entity_attr_names: -- attr names are unique in Entity
  ER!Entity.allInstances()->forAll(e | e.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2))

context ER!Relship inv unique_er_relship_attr_names: -- attr names are unique in Relship
  ER!Relship.allInstances()->forAll(r | r.attrs->forAll(a1,a2 | a1.name = a2.name implies a1=a2))

context ER!Entity inv exist_er_entity_iskey: -- Entity have a key
  ER!Entity.allInstances()->forAll(e | e.attrs->exists(a | a.isKey))

----------------------------------------------
-- POSTCONDITION OF REL

context REL!RELSchema inv unique_rel_schema_names: -- unique name of RELSchemas
  REL!RELSchema.allInstances()->forAll(r1,r2| r1<>r2 implies r1.name<>r2.name)

context REL!RELSchema inv unique_rel_relation_names: -- relation names are unique in RELSchema
  REL!RELSchema.allInstances()->forAll(s | s.relations->forAll(r1,r2| r1<>r2 implies r1.name<>r2.name))

context REL!RELRelation inv unique_rel_attribute_names: -- attribute names unique in RELRelation
  REL!RELRelation.allInstances()->forAll(r | r.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2))

context REL!RELRelation inv exist_rel_relation_iskey: -- RELRelations have a key
  REL!RELRelation.allInstances()->forAll(r | r.attrs->exists(a | a.isKey))
```

B.1.4 Analysis

As discussed in Section 4.4, 3 postconditions of *ER2REL* transformation are verified (i.e. *unique_rel_schema_names*, *unique_rel_relation_names*, *exist_rel_relation_iskey*).

The postcondition *unique_rel_attribute_names* is not verified. We briefly analyse the reason for its failure in this section. By looking at the *ER2REL* transformation, we identify that the *RELAttributes* of *RELRelation* are generated by three ATL rules, i.e. *EA2A*, *RA2A*, *RA2AK*. Each rule initialises the name of the generated *RELAttribute* using different resolved values, i.e. the name of *Entity*'s *ERAttribute* (by *EA2A*), the name of *Relship*'s *ERAttribute* (by *RA2A*), the name of *RelshipEnd*'s *Entity*'s *ERAttribute* (by *RA2AK*).

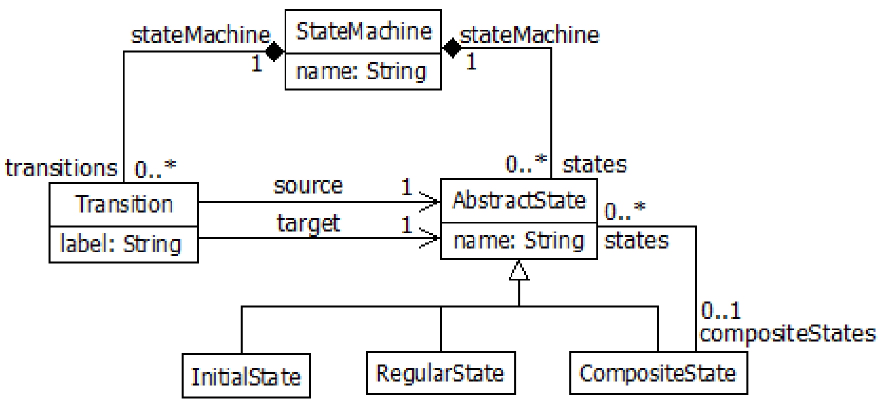
Therefore, we cannot establish *unique_rel_attribute_names* with the given preconditions *unique_er_entity_attr_names* and *unique_er_relship_attr_names* (the other preconditions are irrelevant to *unique_rel_attribute_names*):

- We do not know that the name of *Entity*'s *ERAttribute* and the name of *Relship*'s *ERAttribute* are disjoint.

- It is possible that the same *Entity* is matched by *EA2A* rule and *RA2AK* rule. When that happens, the name of such an *Entity* would be used to initialise the name of *RELAttribute* generated by both rules. As a result, at least two *RELAttribute* will have the same name.

B.2 HSM2FSM Transformation in ATL

B.2.1 State Machine Metamodels (HSM and FSM)



B.2.2 ATL Transformation for HSM2FSM

```
module HSM2FSM;

create OUT : FSM from IN : HSM;

rule SM2SM {
   from sm1 : HSM!StateMachine
   to sm2 : FSM!StateMachine ( name <- sm1.name) }

rule RS2RS {
   from rs1 : HSM!RegularState
   to rs2 : FSM!RegularState ( name <- rs1.name,stateMachine <- rs1.stateMachine ) }

-- Initial states of composite states become regular states in the flattened SM
rule IS2IS {
   from is1 : HSM!InitialState (is1.compositeState.oclIsUndefined())
   to   is2 : FSM!InitialState ( stateMachine <- is1.stateMachine, name <- is1.name ) }

-- Initial states of composite states become regular states in the flattened SM
rule IS2RS {
   from is1 : HSM!InitialState (not is1.compositeState.oclIsUndefined())
   to   is2 : FSM!RegularState ( stateMachine <- is1.stateMachine, name <- is1.name ) }

-- Transitions between two non-composite states are mapped one-to-one
rule T2TA {
   from t1 : HSM!Transition ( not t1.source.oclIsTypeOf(HSM!CompositeState) and
                              not t1.target.oclIsTypeOf(HSM!CompositeState))
   to   t2 : FSM!Transition ( label <- t1.label, stateMachine <- t1.stateMachine,
                              source <- t1.source, target <- t1.target ) }

-- This rule resolves a transition originating from a composite state
rule T2TB {
   from t1 : HSM!Transition,
        src : HSM!CompositeState,
        trg : HSM!AbstractState,
        c : HSM!AbstractState ( t1.source = src and t1.target = trg and c.compositeState = src and
                                not trg.oclIsTypeOf(HSM!CompositeState))
   to   t2 : FSM!Transition ( label <- t1.label, stateMachine <- t1.stateMachine,
                              source <- c, target <- trg ) }

-- This rule resolves a transition ending in a composite state
rule T2TC {
   from t1 : HSM!Transition,
        src : HSM!AbstractState,
        trg : HSM!CompositeState,
        c : HSM!InitialState ( t1.source = src and t1.target = trg and c.compositeState = trg and
                               not src.oclIsTypeOf(HSM!CompositeState) )
   to   t2 : FSM!Transition ( label <- t1.label, stateMachine <- t1.stateMachine,
                              source <- src, target <- c ) }
```

B.2.3 OCL Contracts for HSM2FSM

```
-- PRECONDITION OF HSM

context HSM!StateMachine inv unique_hsm_sm_names: -- different state machines have different names
 HSM!StateMachine.allInstances()->forAll(s1,s2 | s1<>s2 implies s1.name<>s2.name)

context HSM!StateMachine inv unique_hsm_state_names: -- states have unique names
 HSM!AbstractState.allInstances()->forAll(s1,s2 | s1<>s2 implies s1.name<>s2.name);

context HSM!AbstractState inv hsm_state_multi_lower: -- a state does belong to at least one state machine
 HSM!AbstractState.allInstances()->forAll(s | not s.stateMachine.oclIsUndefined())

context HSM!AbstractState inv hsm_state_multi_upper: -- a state does belong to at most one state machine
 HSM!AbstractState.allInstances()->forAll(s |
   HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
     s.stateMachine=sm1 and s.stateMachine=sm2 implies sm1=sm2))

context HSM!AbstractState inv hsm_transition_composite_upper: -- at most one compositeState
 HSM!AbstractState.allInstances()->forAll(c1,c2 |
   c1.oclIsTypeOf(HSM!CompositeState) and c2.oclIsTypeOf(HSM!CompositeState) implies c1=c2)

context HSM!Transition inv hsm_transition_multi_lower: -- a transition does belong to at least one state machine
 HSM!Transition.allInstances()->forAll(t | not t.stateMachine.oclIsUndefined())

context HSM!Transition inv hsm_transition_multi_upper: -- a transition does belong to at most one state machine
 HSM!Transition.allInstances()->forAll(t |
   HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
     t.stateMachine=sm1 and t.stateMachine=sm2 implies sm1=sm2))

context HSM!Transition inv hsm_transition_src_multi_lower: -- a transition has at least one source
 HSM!Transition.allInstances()->forAll(t | not t.source.oclIsUndefined())

context HSM!Transition inv hsm_transition_src_multi_upper: -- a transition has at most one source
 HSM!Transition.allInstances()->forAll(t |
   HSM!AbstractState.allInstances()->forAll(s1,s2 |
     t.source=s1 and t.source=s2 implies s1=s2))

context HSM!Transition inv hsm_transition_trg_multi_lower: -- a transition has at least one target
 HSM!Transition.allInstances()->forAll(t | not t.target.oclIsUndefined())

context HSM!Transition inv hsm_transition_trg_multi_upper: -- a transition has at most one target
 HSM!Transition.allInstances()->forAll(t |
   HSM!AbstractState.allInstances()->forAll(s1,s2 |
     t.target=s1 and t.target=s2 implies s1=s2))

context HSM!Transition inv hsm_transition_src_contain_sm: -- transition and source in the same state machine
 HSM!Transition.allInstances()->forAll(t |
   HSM!AbstractState.allInstances()->forAll(s |
     t.source=s implies HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
       t.stateMachine=sm1 and s.stateMachine=sm2 implies sm1=sm2)))

context HSM!Transition inv hsm_transition_trg_contain_sm: -- transition and target in the same state machine
 HSM!Transition.allInstances()->forAll(t |
   HSM!AbstractState.allInstances()->forAll(s |
     t.target=s implies HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
       t.stateMachine=sm1 and s.stateMachine=sm2 implies sm1=sm2)))

context HSM!Transition inv hsm_transition_contain_sm: -- source and target in the same state machine
 HSM!Transition.allInstances()->forAll(t |
   HSM!AbstractState.allInstances()->forAll(s1,s2 |
     t.source=s1 and t.target=s2 implies HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
       s1.stateMachine=sm1 and s2.stateMachine=sm2 implies sm1=sm2)))
```

```
-- POSTCONDITION OF FSM

context FSM!StateMachine inv unique_fsm_sm_names: -- different target state machines have different names
 FSM!StateMachine.allInstances()->forAll(s1,s2 | s1<>s2 implies s1.name<>s2.name)

context FSM!StateMachine inv unique_fsm_state_names: -- states have unique names
 FSM!AbstractState.allInstances()->forAll(s1,s2 | s1<>s2 implies s1.name<>s2.name);


context FSM!AbstractState inv fsm_state_multi_lower: -- a state does belong to at least one state machine
 FSM!AbstractState.allInstances()->forAll(s | not s.stateMachine.oclIsUndefined())

context FSM!AbstractState inv fsm_state_multi_upper: -- a state does belong to at most one state machine
 FSM!AbstractState.allInstances()->forAll(s |
   FSM!StateMachine.allInstances()->forAll(sm1,sm2 |
     s.stateMachine=sm1 and s.stateMachine=sm2 implies sm1=sm2))

context FSM!Transition inv fsm_transition_multi_lower: -- a transition does belong to at least one state machine
 FSM!Transition.allInstances()->forAll(t | not t.stateMachine.oclIsUndefined())

context FSM!Transition inv fsm_transition_multi_upper: -- a transition does belong to at most one state machine
 FSM!Transition.allInstances()->forAll(t |
   FSM!StateMachine.allInstances()->forAll(sm1,sm2 |
     t.stateMachine=sm1 and t.stateMachine=sm2 implies sm1=sm2))

context FSM!Transition inv fsm_transition_src_multi_lower: -- a transition has at least one source
 FSM!Transition.allInstances()->forAll(t | not t.source.oclIsUndefined())

context FSM!Transition inv fsm_transition_src_multi_upper: -- a transition has at most one source
 FSM!Transition.allInstances()->forAll(t |
   FSM!AbstractState.allInstances()->forAll(s1,s2 |
     t.source=s1 and t.source=s2 implies s1=s2))

context FSM!Transition inv fsm_transition_trg_multi_lower: -- a transition has at least one target
 FSM!Transition.allInstances()->forAll(t | not t.target.oclIsUndefined())

context FSM!Transition inv fsm_transition_trg_multi_upper: -- a transition has at most one target
 FSM!Transition.allInstances()->forAll(t |
   FSM!AbstractState.allInstances()->forAll(s1,s2 |
     t.target=s1 and t.target=s2 implies s1=s2))

context FSM!Transition inv fsm_transition_src_contain_sm: -- transition and source in the same state machine
 FSM!Transition.allInstances()->forAll(t |
   FSM!AbstractState.allInstances()->forAll(s |
     t.source=s implies HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
       t.stateMachine=sm1 and s.stateMachine=sm2 implies sm1=sm2)))

context FSM!Transition inv fsm_transition_trg_contain_sm: -- transition and target in the same state machine
 FSM!Transition.allInstances()->forAll(t |
   FSM!AbstractState.allInstances()->forAll(s |
     t.target=s implies HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
       t.stateMachine=sm1 and s.stateMachine=sm2 implies sm1=sm2)))

context FSM!Transition inv fsm_transition_contain_sm: -- source and target in the same state machine
 FSM!Transition.allInstances()->forAll(t |
   FSM!AbstractState.allInstances()->forAll(s1,s2 |
     t.source=s1 and t.target=s2 implies HSM!StateMachine.allInstances()->forAll(sm1,sm2 |
       s1.stateMachine=sm1 and s2.stateMachine=sm2 implies sm1=sm2)))
```

B.2.4 Analysis

As discussed in Section 4.4, 12 postconditions of *HSM2FSM* transformation are verified except the postcondition *fsm_transition_src_multi_lower*. We briefly analyse the reason for its failure in this section.

By looking at the *HSM2FSM* transformation, we identify that the *source* of *FSM!Transition* is initialised by three ATL rules, i.e. *T2TA*, *T2TB*, *T2TC*.

Each rule initialises the *source* of the generated *FSM!Transition* using different resolved values, i.e. the resolved result of *t1.source* (by *T2TA*), the resolved result of *c* (by *T2TB*), the resolved result of *src* (by *T2TC*). In order to establish that the multiplicity of the *source* of the generated *FSM!Transition* has a lower bound of one, we have to ensure these resolved result are non-empty.

The rules *T2TA* and *T2TC* can establish this easily, since *t1.source* (by *T2TA*) and *src* (by *T2TC*) are known to be not of type *HSM!CompositionState* (as shown by the rule guard). Thus, we know they are either of type *HSM!RegularState* or of type *HSM!InitialState*, and their resolved result will be non-empty (i.e. generated by either rule *RS2RS* or *IS2IS* or *IS2RS*).

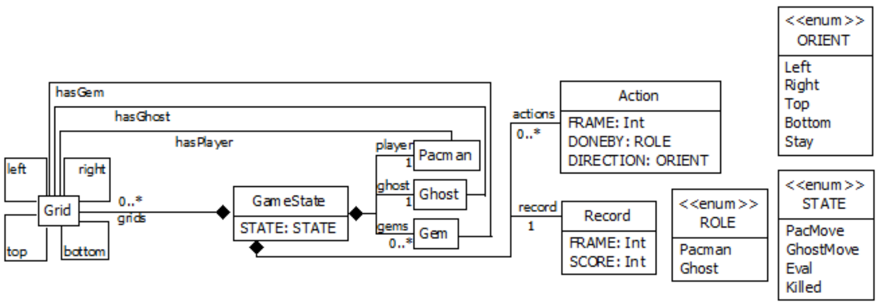
The problem stems from the rule *T2TB*, where we cannot ensure the type of *c* is not *HSM!CompositionState*. Büttner et al. uses the precondition *hsm_transition_composite_-upper* (i.e. at most one *HSM!CompositionState*) to convey this fact. Then, combining with the guard (i.e. *c.compositeState=src* and the fact that *src* is type of *HSM!CompositionState*), they can deduce the type of *c* is not *HSM!CompositionState*.

However, since ATL does not enforce injective matching (Section 6.2.2), *src* and *c* can match the same source model element of type *HSM!CompositionState*. Moreover, there is no ATL rule to process *HSM!CompositionState*. Thus, the resolved result of *c* in this case will be empty. This is why VeriATL reports that the postcondition *fsm_transition_src_-multi_lower* is not verified.

Appendix C

Pacman Transformation in SimpleGT

C.1 Pacman Metamodel



C.2 SimpleGT Transformation for Pacman

```
module Pacman;

rule Collect{
   from
      s : P!GameState(STATE=~Eval,record=~rec),rec : P!Record,pac: P!Pacman,
      gem: P!Gem,grid : P!Grid(hasPlayer=~pac, hasGem=~gem)
   to
      s : P!GameState(STATE=~Eval,record=~recNew),grid : P!Grid(hasPlayer=~pac),
      pac: P!Pacman,recNew: P!Record(FRAME=~rec.FRAME, SCORE=~rec.SCORE+100)
}

rule Kill{
   from
      s : P!GameState(STATE=~Eval),ghost: P!Ghost,pac : P!Pacman,
      grid : P!Grid (hasPlayer=~pac, hasEnemy=~ghost)
   to
      s: P!GameState(STATE=~Killed),ghost: P!Ghost,grid : P!Grid (hasEnemy=~ghost)
}

rule UpdateFrame{
   from
      s : P!GameState(STATE=~Eval,record=~rec),rec : P!Record,pac : P!Pacman
   to
      s: P!GameState(STATE=~PacMove,record=~recNew),pac : P!Pacman,
      recNew: P!Record(FRAME=~rec.FRAME+1, SCORE=~rec.SCORE)
}
```

```
rule PlayerMoveLeft{
   from
      s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid,grid1: P!Grid(hasPlayer=~pac, left=~grid2),
      act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Left)
   not grid2: P!Grid(hasEnemy=~ghost), ghost: P!Ghost
   to
      s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid(hasPlayer=~pac),grid1: P!Grid(left=~grid2)
}

rule PlayerMoveRight{
   from
      s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid,grid1: P!Grid(hasPlayer=~pac, right=~grid2),
      act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Right)
   not grid2: P!Grid(hasEnemy=~ghost), ghost: P!Ghost
   to
      s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid(hasPlayer=~pac),grid1: P!Grid(right=~grid2)
}

rule PlayerMoveTop{
   from
      s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid,grid1: P!Grid(hasPlayer=~pac, top=~grid2),
      act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Top)
   not grid2: P!Grid(hasEnemy=~ghost), ghost: P!Ghost
   to
      s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid(hasPlayer=~pac),grid1: P!Grid(top=~grid2)
}

rule PlayerMoveBottom{
   from
      s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid,grid1: P!Grid(hasPlayer=~pac, bottom=~grid2),
      act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Bottom)
   not grid2: P!Grid(hasEnemy=~ghost), ghost: P!Ghost
   to
      s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid2: P!Grid(hasPlayer=~pac),grid1: P!Grid(bottom=~grid2)
}

rule PlayerMoveStay{
   from
      s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid1: P!Grid(hasPlayer=~pac),
      act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Stay)
   to
      s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
      grid1: P!Grid(hasPlayer=~pac)
}
```

```
rule ghostMoveLeft{
   from
      s:P!GameState(STATE=~GhostMove,record=~rec), rec: P!Record, ghost: P!Ghost,
      grid2:P!Grid, grid1: P!Grid(hasEnemy=~ghost, left=~grid2),
      act : P!Action(DONEBY=~Ghost, FRAME=~rec.FRAME, DIRECTION=~Left)
   to
      s:P!GameState(STATE=~Eval,record=~rec),rec: P!Record,ghost: P!Ghost,
      grid2: P!Grid(hasEnemy=~ghost), grid1: P!Grid(left=~grid2)
}

rule ghostMoveRight{
   from
      s:P!GameState(STATE=~GhostMove,record=~rec), rec: P!Record, ghost: P!Ghost,
      grid2:P!Grid, grid1: P!Grid(hasEnemy=~ghost, right=~grid2),
      act : P!Action(DONEBY=~Ghost, FRAME=~rec.FRAME, DIRECTION=~Right)
   to
      s:P!GameState(STATE=~Eval,record=~rec),rec: P!Record,ghost: P!Ghost,
      grid2: P!Grid(hasEnemy=~ghost), grid1: P!Grid(right=~grid2)
}

rule ghostMoveTop{
   from
      s:P!GameState(STATE=~GhostMove,record=~rec), rec: P!Record, ghost: P!Ghost,
      grid2:P!Grid, grid1: P!Grid(hasEnemy=~ghost, top=~grid2),
      act : P!Action(DONEBY=~Ghost, FRAME=~rec.FRAME, DIRECTION=~Top)
   to
      s:P!GameState(STATE=~Eval,record=~rec),rec: P!Record,ghost: P!Ghost,
      grid2: P!Grid(hasEnemy=~ghost), grid1: P!Grid(top=~grid2)
}

rule ghostMoveBottom{
   from
      s:P!GameState(STATE=~GhostMove,record=~rec), rec: P!Record, ghost: P!Ghost,
      grid2:P!Grid, grid1: P!Grid(hasEnemy=~ghost, bottom=~grid2),
      act : P!Action(DONEBY=~Ghost, FRAME=~rec.FRAME, DIRECTION=~Bottom)
   to
      s:P!GameState(STATE=~Eval,record=~rec),rec: P!Record,ghost: P!Ghost,
      grid2: P!Grid(hasEnemy=~ghost), grid1: P!Grid(bottom=~grid2)
}

rule ghostMoveStay{
   from
      s:P!GameState(STATE=~GhostMove,record=~rec), rec: P!Record, ghost: P!Ghost,
      grid1: P!Grid(hasEnemy=~ghost),
      act : P!Action(DONEBY=~Ghost, FRAME=~rec.FRAME, DIRECTION=~Stay)
   to
      s:P!GameState(STATE=~Eval,record=~rec),rec: P!Record,ghost: P!Ghost,
      grid1: P!Grid(hasEnemy=~ghost)
}
```

C.3 OCL contracts for Pacman

```
context Pacman!GameState pre ValidBoard: -- any two grids are reachable.
  Pacman!GameState->allInstances()->forAll(g | g.grids->forAll(g1,g2 | reachable(g1,g2)));

context Pacman!GameState pre OneGameState: -- at most one GameState.
  Pacman!GameState.allInstances()->forAll(gs1,gs2 | gs1=gs2);

context Pacman!Record pre OneRecord: -- at most one Record.
  Pacman!Record.allInstances()->forAll(r1,r2 | r1=r2);

context Pacman!Pacman pre OnePacman: -- at most one Pacman.
  Pacman!Pacman.allInstances()->forAll(p1,p2 | p1=p2);

context Pacman!Ghost pre OneGhost: -- at most one Ghost.
  Pacman!Ghost.allInstances()->forAll(g1,g2 | g1=g2);

context Pacman!Grid pre existPacman: -- at least one Pacman.
  Pacman!Grid.allInstances()->
    exists(g|not g.hasPlayer.isOclUndefined() and g.hasPlayer.oclIsKindOf(Pacman!Pacman));

context Pacman!Grid pre existGhost: -- at least one Ghost.
  Pacman!Grid.allInstances()->
    exists(g|not g.hasEnemy.isOclUndefined() and g.hasEnemy.oclIsKindOf(Pacman!Ghost));

-----------------------------------------------

context Pacman!Grid post gemReachable: -- all grids containing a gem must be reachable by Pacman.
  Pacman!Grid.allInstances()->forAll(g1,g2|not g1.hasPlayer.isOclUndefined()
    and not g2.hasGem.isOclUndefined() implies reachable(g1,g2));

context Pacman!GameState post PacmanSurvive: -- exists a path where the ghost never kills Pacman.
  Pacman!GameState->allInstances()->forAll(g |
   g.STATE=GhostMove implies g.grids->forAll(g1|
     g1.hasEnemy.oclIsKindOf(Pacman!Ghost) implies not g1.hasPlayer.oclIsKindOf(Pacman!Pacman)));

context Pacman!Action post PacmanMoved: -- the Pacman must move within a time interval I.
  let acts:Sequence(Pacman!Action) = Pacman!Action.allInstances()->select(a|
   a.DONEBY=Pacman and not a.Direction=Stay)->asSequence() in
      Integer.allInstances->forAll(i|
         0<=i<acts->size()-1 implies acts->at(i+1).FRAME-acts->at(i).FRAME<=I);
```

Appendix D

Additional Translational Semantics of the EMFTVM Language

ASM Instruction (S)	Corresponding Boogie Statements ($[\![S]\!]$)
Stack Handling Instructions	
OR *Stmt*	`assert` size(Stk)>0 ; $[\![Stmt]\!]$; Stk := hd(tl(stk)) $\vee$ hd(stk) :: tl(tl(Stk));
XOR	`assert` size(Stk)>1 ; Stk := (hd(stk) $\vee$ hd(tl(stk))) $\wedge$ $\neg$ (hd(stk) $\wedge$ hd(tl(stk))) :: tl(tl(Stk));
IMPLIES *Stmt*	`assert` size(Stk)>0 ; $[\![Stmt]\!]$; Stk := !(hd(tl(Stk))) $\vee$ hd(Stk) :: tl(tl(Stk));
ISNULL	`assert` size(Stk)>0 ; Stk := (hd(Stk)==null) :: tl(Stk);
GET_CB *Stmt*	label$^{\#}$: $[\![Stmt]\!]$
GET_TRANS	not support
Control Flow Instructions	
IFTE $Stmt_1$ $Stmt_2$	`var` cond$^{\#}$: bool ; `assert` size(Stk) $>$ 0 ; cond$^{\#}$:= hd(Stk); Stk := tl(Stk) ; `if` (cond$^{\#}$) $[\![Stmt_1]\!]$ `else` $[\![Stmt_2]\!]$
INVOKE_CB *Stmt n*	$let\ \overline{args} = tk(Stk,n) in$ `assert` size(Stk) $>=$ n ; Stk := args::Stk; $[\![Stmt]\!]$ Stk := hd(Stk)::dp(tl(Stk), n) ;
INVOKE_CB_S *n*	$let\ \overline{args} = tk(Stk,n) in$ `assert` size(Stk) $>=$ n ; Stk := args::Stk; goto label$^{\#}$; Stk := hd(Stk)::dp(tl(Stk), n) ;
INVOKE_STATIC *sig n*	$let\ \overline{args} = tk(Stk,n),\ in$ `var` result$^{\#}$: T ; `assert` size(Stk) $>=$ n ; `call` $result^{\#}$:= invoke(reflect_emftvm(*sig*), $\overline{args}$) ; Stk := result$^{\#}$::dp(Stk, n) ;
INVOKE_SUPER	not support
INVOKE_ALL_CBS	not support
Model Handling Instructions	
NEW *mm*	let cl = hd(Stk) in $let\ clazz = resolve(mm,cl)\ in$: `var` $r^{\#}$: Ref ; `havoc` $r^{\#}$; `assume` $r^{\#} \neq null \wedge \neg$read(heap, $r^{\#}$, $alloc$) ; `assert` size(Stk) $>$ 0 ; `assume` typeof($r^{\#}$) = *clazz* ; heap := update(heap,$r^{\#}$,*alloc*,*true*); Stk := $r^{\#}$::tl(Stk) ;
MATCH	not support
MATCH_S	not support
FIND_TYPE *mm cl*	`assert` size(Stk) $>$ 0 ; Stk := resolve(*mm*, *cl*)::Stk ;
ALLINST_IN *heap cl*	`var` col$^{\#}$: Seq ref ; `assert` size(Stk) $>$ 1 ; col$^{\#}$:= OCLType#allInstance(heap,cl); Stk := col$^{\#}$::tl(tl(Stk)) ;
GET_STATIC *f*	let cl = hd(Stk) in `assert` size(Stk) $>$ 0 ; Stk := read(heap,toRef(cl),*f*)::tl(Stk) ;
SET_STATIC *f*	let cl = hd(tl(Stk)), v = hd(Stk) in `assert` size(Stk) $>$ 1 ; `if` (isCollection(f)) { heap := update(heap,read(heap,toRef(cl),*f*),read(heap,toRef(cl),*f*) $\cup$ *v*); } `else` { heap := update(heap,toRef(cl),*f*,*v*) ; } Stk := tl(tl(Stk));
GET_SUPER	not support
GETENVTYPE	Stk := dtype(ASM)::Stk ;

Appendix E

VeriMTLr in Alternative IVLs

The two most widely used IVLs are Boogie [7], and Why3 [52]. Both of these languages are based on FOL with polymorphic types, and have mature implementations to parse, type-check, and analyse programs. We concentrate on Boogie in this research, but we believe all results can be reproduced in Why3, or other IVLs with comparable functionality. A coarse semantic mapping between Boogie and Why3 is shown in Table E.1, with a focus on the Boogie constructs used in this thesis (Boogie and Why3 constructs are shown in `font` and FONT respectively).

		`Boogie`	WHY3
Type	built-in	`int, bool, map`	INT, BOOL, TUPLE
	user-defined	`type`	TYPE
Function	declaration	`function`	FUNCTION/PREDICATE/INDUCTIVE
Axioms	declaration	`axiom`	AXIOM
	trigger	`e1,e2`	[E1|E2]
Const	declaration	`const`	CONSTANT
Expression	arithmetic	`+,-,*,div,mod`	+,-,*,DIV,MOD
	relation	`>,<,>=,>=,==,!=,<:`	>,<,>=,>=,==,!=,N/A
	binary connector	&&, $\|\|$, $\Leftrightarrow$, $\Rightarrow$	&&, $\|\|$, $\Leftrightarrow$, $\Rightarrow$
	unary connector	`!, old`	!, OLD
	lamda	`lamda`	FUN
	quantifier	`forall, exists`	FORALL, EXISTS
Variable	declaration	`var`	VAR
Procedure	declaration	`procedure`	VAL
	precondition	`requires`	REQUIRES
	postcondition	`ensures`	ENSURES
	modification	`modifies`	WRITES
Implementation	declaration	`implementation`	LET
Statement	label	`id:s`	'ID:S
	assert	`assert (e)`	ASSERT E
	assume	`assume (e)`	ASSUME E
	assign	`id:=e`	ID<-E
	sequencing	`s1;s2`	S1;S2
	goto	`goto id`	N/A
	havoc	`havoc v`	N/A
	if	`if (g) s1 else s2`	IF G THEN S1 (ELSE S2)
	while	`while(g)invs`	WHILE G DO INV S DONE
	call	`call mtd(args)`	(MTD ARGS)

Table E.1 Semantic mapping between Boogie and Why3