Formal Verification of Relational Model Transformations using an Intermediate Verification Language

Zheng Cheng
Computer Science Department, Maynooth University, Ireland

Supervised by: Dr. Rosemary Monahan and Dr. James F. Power

1 OUTLINE OF OBJECTIVES

Model-driven engineering (MDE) has been recognised as an effective way to manage the complexity of software development. Model transformation is widely acknowledged as one of the central ingredients of MDE.

Three main paradigms for developing model transformations are the operational, relational and graph-based approaches. Operational model transformations are imperative in style, and focus on imperatively specifying **how** a model transformation should progress. Relational model transformations (MTr) have a "mapping" style, and aim at producing 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-based transformations use a rewriting style, which applies transformation rules recursively until no more matches can be found.

Because of its mapping-style nature, a MTr is generally easier to write and understand than an operational transformation. Both graph-based transformations and relational transformations can be declarative. However, they are essentially different in their rule matching and execution semantics (Czarnecki and Helsen, 2006). In addition, deciding the confluence and termination of graph-based transformations has been proven undecidable (Plump, 2005; Plump, 1998). This extra layer of complexity hinders the applicability of graph-based transformations compared to relational approaches.

Proving the correctness of MTr is my major concern in this work. Here "correctness" means implicit assumptions about the MTr. These assumptions can be made explicitly by MTr developers via annotations, so-called contracts. Next, I will borrow ER2REL MTr as an example to give a concrete idea about what is meant by the correctness of MTr (Büttner et al., 2012).

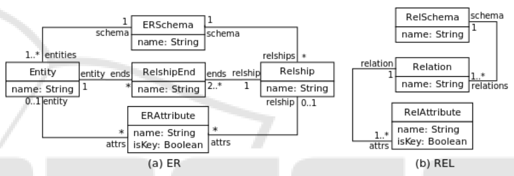

Figure 1: ER and REL metamodels.

The ER2REL MTr transforms from the Entity-Relationship (ER) metamodel (Figure 1(a)) into the RELational (REL) metamodel (Figure 1(b)). Both the ER schema and the REL schema have well-defined semantics. Thus, it is easy to understand their metamodels. The ER2REL MTr is shown in Listing 1. It is written in one of the most widely used MTr languages in industry and academia, namely the ATL transformation language (ATL) (Jouault et al., 2008). The ER2REL MTr is defined via a list of ATL rules in a mapping style. Each rule defines how to map **from** the source model element(s) **to** the target model element(s). A rule is applicable when the source model element(s) fulfils the rule guard. Each rule initialises the attribute/association of the generated target model element via the binding operator ($\leftarrow$). This binding operator performs an implicit resolution algorithm to resolve the right hand side of the operator, and assigns the resolution result to its left hand side (Jouault et al., 2008).

In this work, I am specifically interested in four types of MTr correctness:

1. **Syntactic Correctness.** This concerns whether every well-formed source model is able to generate a well-formed target model. The ER2REL transformation fails this correctness criterion, because the *EA2A* rule (Listing 1) will generate a dangling *RelAttribute* that is not attached to any *Relation* in the target model.
2. **Semantic Correctness.** This concerns, when the source models satisfy the preconditions, whether the postconditions will always hold on the corresponding target models after executing the MTr. For example, if a semantic correctness contract

```
1  module ER2REL; create OUT : REL from IN : ER;
2
3  rule S2S {
4  from s: ER!ERSchema
5  to t: REL!RELSchema (relations <- s.entities, relations
         <- s.relships)}
6
7  rule E2R {
8  from s: ER!Entity
9  to t: REL!Relation ( name<-s.name) }
10
11 rule R2R {
12 from s: ER!Relship
13 to t: REL!Relation ( name<-s.name) }
14
15 rule EA2A {
16 from att: ER!ERAttribute, ent: ER!Entity (att.entity =
         ent)
17 to t: REL!RELAttribute ( name<-att.name,
         isKey<-att.isKey) }
18
19 rule RA2A {
20 from att: ER!ERAttribute, rs: ER!Relship (
         att.relship=rs )
21 to t: REL!RELAttribute ( name<-att.name,
         isKey<-att.isKey, relation<-rs ) }
22
23 rule RA2AK {
24 from att: ER!ERAttribute, rse: ER!RelshipEnd (
         att.entity=rse.entity and att.isKey=true )
25 to t: REL!RELAttribute ( name<-att.name,
         isKey<-att.isKey, relation<-rse.relship )}
```

Listing 1: ER2REL Transformation Specification in ATL.

of ER2REL is given as follows: the precondition specifies that "the *relships* of *ERSchema* have different names", then would the postcondition of "all the *relations* of *RELSchema* have different names" be established by executing the ER2REL MTr shown in Listing 1?

The answer to this question depends on how the execution semantics of the ATL transformation specification is interpreted. More specifically, it depends on how the two bindings to the *relations* associations on line 5 are interpreted. (i) If we interpret that the second binding **overwrites** the effect of first binding, then the postcondition will hold. (ii) However, if we interpret that the second binding **composites** the effect of first binding, then the postcondition will not hold. Unfortunately, which one to choose is not explicitly specified by the existing documentation of ATL.

3. **Semantic Consistency.** This concerns whether the execution semantics of the transformation specification is consistent with the execution semantics of its transformation implementation. For example, by studying the transformation implementation of ATL, binding an association multiple times is a composition process, i.e. interpretation (ii) is consistent with the execution semantics of the transformation implementation. Thus, the previous postcondition does not hold on the ER2REL transformation. If we choose interpretation (i) instead, it will be semantically inconsistent with the underlying transformation implementation. As a result, the semantic correctness of ER2REL is unreliable, i.e. it will erroneously conclude that the postcondition always holds when it might not after MTr execution.
4. **Termination.** This concerns whether executing the transformation specification will terminate. The ER2REL transformation terminates since it does not uses any ATL constructs that can lead to non-termination (Jouault et al., 2008).

For the rest of the paper, I refer to the first two correctness types as the partial correctness of MTr, and the last two as the total correctness of MTr.

Thus, the main objective of this work is to ***design a static verifier for the target MTr language by applying formal methods, which allows the designed verifier to analyse annotated MTr and check that the given correctness contracts are never violated***. 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 verification, since its correctness is guaranteed for all the possible source models.

On top of the main objective, I expect to achieve two sub-objectives in this work:

- Systematically and modularly design the static verifier to ensure the verifier design can be reused for other MTr languages.
- Soundly design the static verifier to ensure the reliability of its verification result.

2 RESEARCH PROBLEM

From the objectives of my work, I identify two research problems.

Research Problem One: My 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.

Research Problem Two: I want to investigate whether the translation validation approach from compiler verification can be automated to ensure the reliability of the verification result, i.e. to be able to check consistency between the execution semantics of each transformation specification and its corresponding transformation implementation.

3 STATE OF THE ART

In this section I primarily focus on the literature that falls within the scope of formal verification of MTr. I categorise them by the formal method applied.

Simulation. Simulation approaches require that a mathematical model be developed. This mathematical model represents the key characteristics of the MTr (e.g. source and target metamodels, the behaviour of the MTr specification). 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 input. For example, colored Petri nets have been used to simulate Query/View/Transform (QVT) MTr. Thus, a colored Petri nets engine can be used to check various contracts such as termination (Wimmer et al., 2009; Guerra and de Lara, 2014). Similarly, Troya and Vallecillo use rewriting logic to simulate ATL MTr in the Maude system (Troya and Vallecillo, 2011). Their product allows a reachability analysis of the ATL.

Model Checking and Model Finding. Similar to simulation approaches, model checking and model finding approaches also require that a mathematical model be developed (from the metamodels and the MTr specification). However, no particular input is needed when the model checking/finding is running.

A subtle difference between the model checking and model finding approaches is in the way that they use the developed mathematical model (Huth and Ryan, 2004). The former one 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 later one finds mathematical models which form counter-examples to the contracts made by the user.

Lúcio et al. develop an off-the-shelf model checker for the DSLTrans language. Their model checker allows the user to check the syntactic correctness (encoded in algebra) of the generated target models (Lúcio et al., 2010; Lúcio and Vangheluwe, 2013). The key to developing the model checker is the expressiveness reduction of the DSLTrans language, i.e. any constructs that might imply unbounded recursion or non-determinism are avoided. Thus, the state space of DSLTrans model transformations is always finite.

Anastasakis et al. have designed the UML2Alloy tool to perform model finding (Anastasakis et al., 2007). The novelty of their work is the use of Alloy, which is a verification language for SAT-based model finding (Jackson, 2006). Anastasakis et al. use Alloy as an intermediate verification language to ease (i) the encoding of MOF metamodels (enriched with syntactic correctness contracts expressed in OCL) and MTr to Alloy; (ii) the generation of SAT formulas from Alloy. Jackson et al. have designed the Formula framework (Jackson et al., 2011), which is based on the Z3 SMT solver (de Moura and Bjørner, 2008). The main contribution is that they systematically encode MOF metamodels and MTr specifications using algebraic data types. The contracts are given by First-Order-Logic (FOL). Consequently, they can use their framework to find models that witness violations of syntactic correctness in the given MTr specification.

Theorem Proving. Theorem proving approaches formalise both the MTr specification and its contracts into formulas. Verification consists of applying deduction rules (of an appropriate logic) to incrementally build the proof.

Combemale et al. present a pen and paper bisimulation proof to show that the ATL MTr generates a Petri nets model that preserves the observational operational semantics of an xSPEM model (Combemale et al., 2009). Calegari et al. encode the ATL MTr and its metamodels into inductive types (Calegari et al., 2011). The contracts for semantic correctness are given by OCL expressions, and 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, there is a line of research which develops the concept of proof-as-model-transformation methodology (Chan, 2006; Poernomo, 2008; Poernomo and Terrell, 2010; Lano et al., 2014). This is the opposite of traditional theorem proving approaches. At its simplest, the idea is to represent the metamodels as terms. Then, each MTr specification and its contracts are encoded together as an $\forall\exists$ type. Next, type theory (for the lambda-calculus) can be regarded as a proof system to verify the encoded $\forall\exists$ type. Finally, a program can be extracted from the proof.

Similar to the proof-as-model-transformation methodology, the UML-RSDS (Reactive System Development Support of UML) is a tool-set for developing correct model transformations by construction (Lano et al., 2014). 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 specification. The specification is optionally constrained by OCL contracts on source and target metamodels. Then, the MTr specification is verified against its contracts by translating both into B AMN (B Abstract Machine Notation) or as input for the Z3 SMT solver for theorem proving. Finally, the verified model transformation design can be synthesised to an executable transformation implementation (such as a Java program or an ATL transformation).

Büttner et al. automate the process of theorem proving by a novel use of SMT solvers (Büttner et al., 2012). 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 semantic correctness) directly into FOL formulas. The formulas represent the execution semantics of the ATL transformation specification, 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.

Summary. Essentially, simulation, model checking and model finding approaches are bounded. This means the MTr specification 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.

Theorem proving approaches are unbounded. Therefore, this is preferable when the user requires that contracts hold for the MTr specification over an infinite domain. However, most of the theorem proving approaches require guidance and expertise from the user (Combemale et al., 2009; Calegari et al., 2011; Chan, 2006; Poernomo, 2008; Poernomo and Terrell, 2010; Lano et al., 2014). This can be ameliorated by a novel use of SMT-solvers such as that presented by Büttner et al (Büttner et al., 2012). However, current usage of SMT-solvers lacks an intermediate form to bridge between the MTr language and the back-end SMT-solver. This compromises the reusability and modularity of the verifier. Alloy has been used as an intermediate verification language for SAT-based model finding (Anastasakis et al., 2007). However, the intermediate verification language has not yet been adopted in theorem proving approaches for MTr.

All the theorem proving approaches I found rely on deriving a formula to represent the execution semantics of MTr specifications. However, the reliability of the derived formulas has not been considered (Ab. Rahim and Whittle, 2014). If the derived formulas incorrectly represent the execution semantics of the MTr specification, then the soundness of the verifier is greatly compromised.

Finally, OCL is one of the most popular language to express the transformation contracts (Anastasakis et al., 2007; Lano et al., 2014; Calegari et al., 2011; Büttner et al., 2012). However, OCL sometimes can be verbose, and difficult to read/write (Vaziri and Jackson, 2000).

4 METHODOLOGY

To allow automatic theorem proving for MTr, I have designed the VeriMTLr development framework to provide rapid verifier construction for MTr languages. In particular, the VeriMTLr framework enables designing verifiers that perform static verification of partial (i.e. semantic correctness and syntactic correctness) and total (i.e. semantic consistency and termination) correctness. The architecture of my VeriMTLr framework is shown in Figure 2. Next, I will briefly explain the core components that reduce coding costs, time and errors during verifier construction.

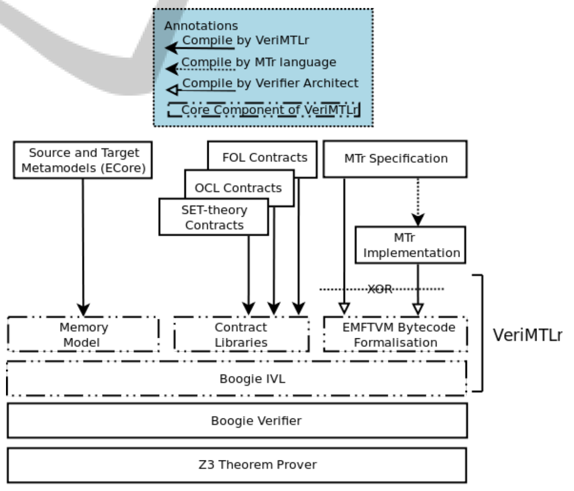

Figure 2: The Architecture of my VeriMTLr Development Framework.

4.1 The Boogie Intermediate Verification Language

The essential idea of the VeriMTLr framework is to formalise the metamodels, the execution semantics of the transformation specification, and the transformation contracts into an IVL, such as Boogie (Barnett et al., 2006) or WHY3 (Filliâtre, 2013). Then, the IVL can interact with its underlying theorem prover(s) for verification. The result from the prover(s) will verify the correctness of the transformation specification with respect to its contracts.

Using an IVL in verifier design has two advantages. First, the formalisation for metamodels, transformation contracts, and even the execution semantics of the transformation specification can be encapsulated in an IVL as libraries. These formalisations are then reusable for design verifiers for different MTr languages. Second, an IVL can bridge between the front-end MTr language and the back-end theorem prover. The benefit here is to focus on generating verification tasks for the front-end language in a structural way, and to delegate the task of interacting with theorem provers to the IVL.

I use the Boogie IVL (Boogie) in the VeriMTLr framework. Boogie is procedure-oriented, and is a barebones implementation of Hoare-logic. It provides imperative statements (such as assignment, goto, if, while and call statements) to implement procedures, and supports FOL contracts (i.e. pre/postconditions) to specify procedures. In addition, Boogie allows the declaration of types, constants, functions and axioms, which are mainly used to encode libraries that define background theories and language properties. A Boogie procedure is verified if its implementation satisfies its contracts. The verification of Boogie procedures is performed by the Boogie verifier, which uses the Z3 SMT solver as its underlying theorem prover.

4.2 A Memory Model for Metamodel Formalisation

Each MTr is defined via mappings from the source metamodel to the target metamodel. Thus, one essential requirement of MTr verification is to decide how to formalise the metamodels. The concepts of metamodelling share many similarities with object oriented (OO) programming language constructs. Thus, I decided to reuse the formalisation of OO programs (specifically its memory model) to formalise metamodels. I provide a metamodel compiler in the VeriMTLr framework, which compiles from the Ecore metamodels directly to the Boogie formalisation based on a particular OO memory model.

The OO memory model I chose uses an updatable array to represent the run-time heap. The array maps memory locations (identified by an object reference and an object field) to values. Memory operations are given by the array operations *store* and *select*. The data type of an object reference is accessed through the *datatype* operation. A class can inherit from another class, but multiple-inheritance is not supported. The major advantage of the chosen memory model is that we are able to quantify over all the memory locations. Thus, it is easier to specify what is not changed in the memory model.

The implications of my formalisation for metamodels are twofold. First, because of the shared similarities with OO constructs, the metamodel formalisation will be easy to comprehend and reuse. Second, this OO memory model has been used by several program verifiers (e.g. Spec# (Barnett et al., 2004), KIV (Stenzel, 2004)). Therefore, it will enhance the interoperability between OO program verifiers and MTr verifiers.

4.3 Contract Libraries

I provide three contract libraries encoded in Boogie, i.e. the OCL library, the SET-theory library and the FOL library. I also provide three compilers to compile OCL, SET and FOL expressions into the corresponding Boogie expressions that are defined in the contract libraries. The goal is to provide flexibility while specifying the transformation contracts.

The development of contract libraries benefits from choosing an IVL in verifier design. For example, my OCL library is built on top of an existing Boogie library from the Dafny verification language (Leino, 2010), and is also made available to others for reuse. The existing one provides the mathematical theories for sets, sequences, bags and maps, which have an intuitive mapping to the OCL collections. On top of these, I further introduce the ordered-set collection data type (with 3 OCL operations), and 6 OCL iterators on the sequence and ordered-set data types (i.e. exists, forall, isUnique, select, collect and reject iterators)[1]. I draw on previous work of Leino and Monahan to guide my axiomatisation for the iterators (Leino and Monahan, 2009).

4.4 EMFTVM Bytecode Formalisation

Generally, when designing the verifier for a MTr language, the verifier architects face two kinds of dilemma. First, if the MTr language does not have a transformation implementation yet, can the architects ensure that the derived execution semantics of the transformation specification is implementable? Second, if the MTr language already has a transformation implementation, can the architects ensure that the derived semantics correctly represent the execution semantics of the underlying transformation implementation? If the answers to these questions are no, the semantic consistency of MTr is in doubt. Therefore, I

[1]The reason that the iterators are not defined on all collection data types is because my Boogie encoding for the set and bag data types is not enumerable.

provide a translational semantics for the bytecode of the EMF transformation virtual machine (EMFTVM) in Boogie, to provide a solution for answering these questions.

EMFTVM is a stack-based VM, which aims at providing a common execution semantics for the transformation implementation of rule-based MTr languages (Wagelaar et al., 2011). It is based on EMF (which represents a de facto standard for modelling today), and uses a low-level bytecode language to implement model transformations. The bytecode language supports 47 different instructions. Apart from the general-purpose instructions for control flow and stack handling, several EMF-specific instructions exist (e.g. SET, GET, DELETE). Existing model transformation languages that target the EMFTVM include ATL and SimpleGT (Wagelaar et al., 2011).

My formalisation of the EMFTVM bytecode serves two purposes. First, it provides a formal documentation of the EMFTVM bytecode. For example, if an MTr language has no transformation implementation yet, my EMFTVM bytecode formalisation assists the verifier architects in designing the transformation implementation correctly. Second, my EMFTVM bytecode formalisation is an interface to Boogie. It allows the verifier architect to map the execution semantics of each transformation implementation (if there is one) into Boogie. After the execution semantics of the corresponding transformation specification has also been mapped to Boogie, semantic consistency between the two can be verified in the same verification language. This verification is based on each pair of transformation specification and implementation. The idea is borrowed from the translation validation technique used in compiler verification (Leroy, 2006). The benefit is that instead of verifying that the transformation specification is always consistent with its transformation implementation (which is difficult to automate), I can automatically verify the consistency of each pair.

My formalisation of the EMFTVM bytecode is based on giving its translational semantics in Boogie, i.e. translating the operational semantics of the EMFTVM bytecode into Boogie. The operational semantics of the EMFTVM bytecode is derived by re-engineering the implementation of the EMFTVM. I have not given any formal proof for my derivation. To my knowledge, I am the first one to introduce the EMFTVM bytecode formalisation. Thus, there is no alternative and commonly-accepted formal semantics for it that allows me to reason against. However, when there is one, I can adapt existing techniques to prove their equivalence (Combemale et al., 2009; Lehner and Müller, 2007; Apt et al., 2009).

5 EXPECTED OUTCOME

I identify three use cases for the VeriMTLr framework.

Verifying an ATL Transformation. VeriATL is the first verifier I designed using the VeriMTLr framework. It is designed for partial (i.e. syntactic correctness and semantic correctness) and total correctness (i.e. termination and semantic consistency) verification of an essential subset of ATL (i.e. match rules). VeriATL is encapsulated in the VeriMTLr framework, and made available for public use.

To perform verification of partial correctness, a user of VeriATL sends the relevent metamodels, an ATL transformation specification and the transformation contracts to VeriATL. These inputs are compiled by VeriATL into Boogie, and then verified for contract violations. The verification result is reported as traces which point to the potential error location in the ATL transformation specification.

To perform verification of semantic correctness, the user of VeriATL will send an ATL transformation specification and its corresponding transformation implementation (i.e. the ATL stack machine implementation) into VeriATL. The result determines whether the transformation implementation is consistent with the ATL transformation specification.

I have successfully applied VeriATL to verify the partial and total correctness of ER2REL transformation shown in Listing 1. Its full description can be found on my on-line repository (Cheng et al., 2013).

Developing a Verifier for a New MTr Language. In order to build a verifier for a new MTr language and to verify partial correctness, the verifier architects can reuse the metamodel formalisations and contract libraries in VeriMTLr. However, they are responsible for deriving the execution semantics of the transformation specification for the new MTr language, and then formalising it into Boogie. They will find that the VeriATL case study is useful in specifying the execution semantics of the ATL transformation specification.

Verifying Termination and Compilation of a MTr Language. In order to verify the termination and the correct compilation of a MTr Language, a verifier that verifies total correctness needs to be built for the MTr language. The architects of such a verifier should build on top of the verifier for partial correctness verification (of the MTr language). Thus, they can reuse the Boogie formalisation for the execution semantics of the transformation specification when building the verifier for total correctness. Next, the architects will map the execution semantics of the transformation implementation (of the MTr language) to my formalisation of the EMFTVM bytecode in the VeriMTLr framework.

6 STAGE OF THE RESEARCH

I have spent three years developing the VeriMTLr framework. The goal is to provide rapid verifier construction for MTr languages. In particular, the VeriMTLr framework assists in designing verifiers that allow automatic theorem proving of partial and total correctness of MTr. It encapsulates four core components to reduce coding costs, time and errors of verifier construction, i.e. the Boogie IVL, the memory model (for formalizing metamodels), the contract libraries (which formalises OCL, SET theory and FOL), and the EMFTVM bytecode formalisation.

Next, I illustrate three identified limitations of the VeriMTLr framework. First, my framework relies on Boogie, which sits on top of the Z3 SMT solver. Z3 is based on first order predicate logic with equality, which restricts the expressiveness of my framework. For example, it is not possible to express transitive closure properties.

Second, the soundness of the VeriMTLr framework depends on the correctness of my formalisations for metamodels, contract libraries and EMF bytecode. The correctness of these formalisations are challenging theoretical problems that require well-defined and commonly accepted formal semantics of each. However, to my knowledge, none of them are currently available. However, my formalisations are encoded in Boogie, which yields intuitive formalisations for inspection.

Third, the completeness of the VeriMTLr framework remains one of my major concerns. The verifier constructed by the VeriMTLr might not be able to verify a model transformation specification against its contracts, even if the two are verifiable. The incompleteness of the VeriMTLr framework might be due to the underlying SMT solver (i.e. the undecidability of first-order-logic). It might also be due to my formalisations. For example, the formalisation of the *sequence* data type in my OCL library only contains the essential definition for *append* operation. The auxiliary axioms such as "any sequence appended with an empty sequence is the original sequence" are not in my formalisation. The decision is made deliberately. Essentially, I reduce each MTr verification problem into a SMT formula solving problem. Extra axioms will generate longer SMT formulae, and might be more difficult to solve. Therefore, I think it is better to present the missing auxiliary axioms as lemmas, which will be introduced on demand. Moreover, presenting only the essential axioms is a strategy that helps manual inspection and reduces the possibility of inconsistent axioms.

In the last year of my research, I plan to work with more MTr scenarios (preferably with transformation contracts) to ensure first order predicate logic with equality is expressive enough for MTr verification in practice.

I have not shown the reusability of the VeriMTLr framework. Therefore, it would be interesting to work with the VeriMTLr to design a verifier for another target MTr language. I expect the core components in my framework can be reused to systematically design a modular and sound verifier for this target MTr language.

REFERENCES

Ab. Rahim, L. and Whittle, J. (2014). A survey of approaches for verifying model transformations. *Software & Systems Modeling*, Pre-Printing.

Anastasakis, K., Bordbar, B., and Küster., J. M. (2007). Analysis of model transformations via Alloy. MODEVVA'07, Workshop on Model-Driven Engineering, Verification and Validation.

Apt, K. R., de Boer, F. S., and Olderog, E.-R. (2009). *Verification of Sequential and Concurrent Programs*. Springer, 3rd edition.

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 *FMCO'06, 4th International Conference on Formal Methods for Components and Objects*. Springer.

Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. (2004). Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3.

Büttner, F., Egea, M., Cabot, J., and Gogolla, M. (2012). On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *MoDELS'12, 15th International Conference on Model Driven Engineering Languages and Systems*. Springer.

Calegari, D., Luna, C., Szasz, N., and Tasistro, Â. (2011). A type-theoretic framework for certified model transformations. In *SBMF'11, 14th Brazilian Symposium on Formal Methods*. Springer.

Chan, K. (2006). Formal proofs for QoS-oriented transformations. In *EDOCW '06, 10th IEEE International Conference Workshops on Enterprise Distributed Object Computing*. IEEE.

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

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).

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal - Model-driven software development*, 45(3).

de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *TACAS'08, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

Filliâtre, J.-C. (2013). One logic to use them all. In *CADE'13, 24th International Conference on Automated Deduction*. Springer.

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).

Huth, M. and Ryan, M. (2004). *Logic in Computer Science*. Cambridge University Press, 2nd edition.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Jackson, E. K., Levendovszky, T., and Balasubramanian, D. (2011). Reasoning about metamodeling with formal specifications and automatic proofs. In *MODELS'11, 14th International Conference on Model Driven Engineering Languages and Systems*. Springer.

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

Lano, K., Clark, T., and Kolahdouz-Rahimi, S. (2014). A framework for model transformation verification. *Formal Aspects of Computing*, Pre-Printing.

Lehner, H. and Müller, P. (2007). Formal translation of bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science*, 190(1).

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *LPAR'10, 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Springer.

Leino, K. R. M. and Monahan, R. (2009). Reasoning about comprehensions with first-order SMT solvers. In *SAC '09, ACM Symposium on Applied Computing*. ACM.

Leroy, X. (2006). Formal certification of a compiler backend or: Programming a compiler with a proof assistant. *SIGPLAN Notices*, 41(1).

Lúcio, L., Barroca, B., and Amaral, V. (2010). A technique for automatic validation of model transformations. In *MoDELS'10, 13th International Conference on Model Driven Engineering Languages and Systems*. Springer.

Lúcio, L. and Vangheluwe, H. (2013). Model transformations to verify model transformations. VOLT'13, Workshop on Verification And Validation Of Model Transformations.

Plump, D. (1998). Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2).

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

Poernomo, I. (2008). Proofs-as-model-transformations. In *ICMT'08, 1st International Conference on Model Transformation*. Springer.

Poernomo, I. and Terrell, J. (2010). Correct-by-construction model transformations from partially ordered specifications in Coq. In *ICFEM'10, 12th International Conference on Formal Engineering Methods*. Springer.

Stenzel, K. (2004). A formally verified calculus for full Java card. In *Algebraic Methodology and Software Technology*, volume 3116. Springer.

Troya, J. and Vallecillo, A. (2011). A rewriting logic semantics for ATL. *Journal of Object Technology*, 10.

Vaziri, M. and Jackson, D. (2000). Some shortcomings of OCL, the object constraint language of UML. In *TOOLS '00, 34th Technology of Object-Oriented Languages and Systems*. IEEE.

Wagelaar, D., Tisi, M., Cabot, J., and Jouault, F. (2011). Towards a general composition semantics for rule-based model transformation. In *MoDELS'11, 14th International Conference on Model Driven Engineering Languages and Systems*. Springer.

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. DSM'09, 9th OOPSLA Workshop on Domain-Specific Modeling.