Verifying SimpleGT Transformations Using an Intermediate Verification Language

Zheng Cheng, Rosemary Monahan, and James F. Power

Computer Science Department, Maynooth University, Co. Kildare, Ireland
{zcheng, rosemary, jpower}@cs.nuim.ie

Abstract. Previously, we have developed the VerMTLr framework that allows rapid verifier construction for relational model transformation languages. VerMTLr draws on the Boogie intermediate verification language to systematically design a modular and reusable verifier. It also includes a modular formalisation of EMFTVM bytecode to ensure verifier soundness. In this work, we will illustrate how to adapt VerMTLr to design a verifier for the SimpleGT graph transformation language, which allows us to soundly prove the correctness of graph transformations. An experiment with the Pacman game demonstrates the feasibility of our approach.

Keywords: model transformation verification, graph transformation, SimpleGT, automatic theorem proving, intermediate verification language, Boogie

1 Introduction

Relational model transformation (MTr) is one of the main paradigms used in model transformation (MT). It has a "mapping" style, and aims at producing a declarative transformation specification that documents what the model transformation intends to do. Graph transformation (GT) is another model transformation paradigm. It has a rewriting style, and usually represents 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 behaviours 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.

In this work, we will focus on GTs. SimpleGT is a textual GT language based on double push-out semantics [17]. 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 the Object Constraint Language (OCL) for both its data types and its declarative expressions. Then, the SimpleGT program is compiled into an EMFTVM implementation to be executed.

Verifying the correctness of a SimpleGT transformation means proving assumptions about the SimpleGT program. These assumptions can be made explicitly by transformation developers via annotations, so-called contracts. The contracts are usually expressed in OCL because of its declarative and logical nature.

To allow automatic correctness verification for MTr, we have designed the VeriMTLr development framework to provide rapid verifier construction [7]. At the core of our framework is the Boogie intermediate verification language (Boogie) which enables Hoare-logic-based automatic theorem proving [4]. Boogie provides imperative statements (such as assignment, if and while statements) to implement procedures, and supports first-order-logic (FOL) contracts (i.e. pre/-postconditions) to specify procedures. It allows type, constant, function and axiom declarations, which are mainly used to encode libraries that define data structures, 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[1] as its underlying theorem prover.

Our framework encapsulates the semantics of the EMF metamodel library (based on the Burstall-Bornat memory model [6]) and a subset of OCL (i.e. OCLAny, OCLType, Primitive and collections) as Boogie libraries. The two libraries provide a foundation to encode the execution semantics of MTr languages. The unique feature of VeriMTLr is its ability to ensure sound verifier design through a translation validation approach. That is, it automatically verifies that each encoded execution semantics of an MTr specification is sound with respect to its corresponding runtime behaviour of the MTr implementation [9].

Our main contribution in this work is to articulate how to adapt VeriMTLr to soundly design the VeriGT system, which is used to verify the correctness of SimpleGT transformations. In particular:

– We demonstrate the difference between the execution semantics of relational and graph transformations, and quantify how the difference would affect their verifier design (Section 3).
– We demonstrate VeriGT on the wellknown Pacman game, and share our experience on verifying three contracts for the Pacman game (Section 4). One interesting observation is that by carefully designing the Pacman metamodel, we do not require explicit tool-support or formalisms (e.g. CTL) to handle temporal constraints.

2 The SimpleGT Language and its Correctness

We use the Pacman game adapted from [15] to introduce the SimpleGT language. The game is based on the Pacman metamodel as shown in Fig. 1. The game consists of a single *Pacman*, a *ghost* and zero or more *gems* on a game board (consisting 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.

We have defined the semantics of a Pacman game via 13 GT rules in SimpleGT (Fig. 2). Each rule includes an input pattern ($from$ section), a correspon-

[1] Z3. http://z3.codeplex.com/.

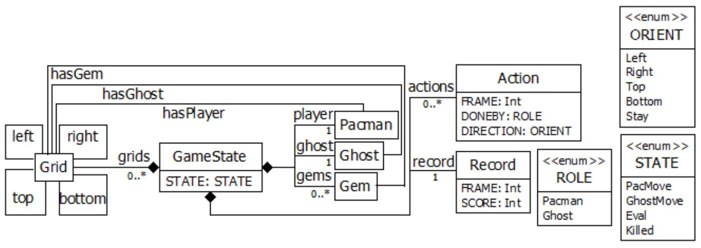

Fig. 1: Pacman metamodel

dence 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 differences from input pattern to correspondence pattern are deleted, the correspondence pattern is left unchanged, and the differences from output pattern to correspondence pattern are created. SimpleGT uses explicit negative application condition patterns (NACs), which specify input patterns that prevent the rule from matching. Optionally, the matching operator ('=∼') can be used to match the existence of an edge or an attribute value in the input or output pattern. In addition, SimpleGT enforces injective matching and follows an automatic "fall-off" rule scheduling, for which we give more details in Section 3.

```
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) } ...
```

Fig. 2: Part of the Pacman graph transformation rules in SimpleGT

We have 10 rules to move Pacman and the ghost in different directions (5 rules for each role). We prevent Pacman from committing suicide by prohibiting it from moving to the grid that already has the ghost. 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.

Our quest is to systematically design a modular and reusable verifier that verifies the correctness of the SimpleGT programs. The correctness of a SimpleGT program is specified using OCL contracts. In this work, we specify three contracts as shown in Fig. 3, i.e. *gemReachable*, *PacmanSurvive* and *PacmanMoved*. The rationale behind each specified contract is explained further in Section 4. 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 4).

```
context GameState pre ValidBoard: −− any two grids are reachable.
  self.grids->forAll(g1,g2:Grid|reachable(g1,g2))
... −− other well−formatness contracts of the Pacman game.
context Grid inv gemReachable: −− all grids containing a gem must be reachable by Pacman.
  self.allInstances()->forall(g1,g2:Grid|not g1.hasPlayer.isOclUndefined() and not
        g2.hasGem.isOclUndefined() implies reachable(g1,g2))
context GameState inv PacmanSurvive: −− exists a path where the ghost never kills Pacman.
  self.STATE==GhostMove implies self.grids->forall(g1:Grid|g1.hasEnemy.oclIsKindOf(Ghost)
        implies not g1.hasPlayer.oclIsKindOf(Pacman))
context Action inv PacmanMoved: −− the Pacman must move within a time interval I.
  let col:Sequence=self.allInstances()->select(a:Action|a.DONEBY=Pacman and not
        a.Direction=Stay)->asSequence() in
    col->forall(i:int|0<=i<col->size()-1 implies col->at(i+1).FRAME-col->at(i).FRAME<=I)
```

Fig. 3: OCL contracts for Pacman

3 Design of VeriGT Verifier

The three core components of the VeriGT design are the Boogie encoding for the EMF metamodel, the OCL transformation contracts and the execution semantics of SimpleGT. We can directly reuse the EMF metamodel library and OCL library from our previous work on the VeriMTLr framework for the first two kinds of encoding. The main challenge stems from encoding the execution semantics of SimpleGT. Specifically, the semantics of rule scheduling in SimpleGT must be able to match rules with their own output, i.e. re-matching after each rule application[2]:

– 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. no more match for any rules), the rule scheduling stops.

To the best of our knowledge, none of MTr languages share the same execution semantics. Taking the ATL MTr language as an example, we found three major differences between the execution semantics of ATL and SimpleGT:

– First, ATL is scheduled to first match each rule, and then apply each rule. This is to ensure the confluence of an ATL transformation [3].
– Second, ATL applies an implicit resolution algorithm while binding the target metamodel elements, which do not exist in SimpleGT.
– Third, ATL has a simpler *match step*, whereas the *match step* of SimpleGT is more complex:
  • The first sub-step performs a structural pattern matching (by applying a search plan strategy [16]), 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

[2] For simplicity we do not consider rule inheritance [17].

structural pattern are unique. ATL does not enforce the same constraint on matching.
- 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. constraints on the attributes of model elements given by the matching operator).

Thus, we cannot reuse our encoding of the execution semantics for MTr languages here. In Fig. 4 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. 3 as follows:

– First, the OCL contracts are encoded as a Boogie contract (line 1 - 11). For instance, the contract *PacmanSurvive* of Fig. 3 is encoded as both a precondition (line 3 - 5) and a postcondition (line 9 - 11).
– Then, the execution semantics of the Pacman SimpleGT program is encoded as a Boogie implementation (line 13 - 29). Specifically, the rule scheduling is encoded in a loop (line 15 - 27). During the loop, the execution semantics of *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 23). The fall-off-matching is repeated until the last GT rule jumps out of 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.

4 Evaluation

We evaluate VeriGT on the Pacman game, previously presented by [15]. Our evaluation runs on an Intel 2.93 GHz machine with 4 GB of memory running on Windows OS. Verification times are recorded in seconds. Table. 1 shows the performance of our transformation correctness verification. The *second* column shows the size of the Boogie code generated to verify each of the transformation contracts that are specified in Fig. 3 (including Boogie encoding for the Pacman metamodel, transformation contract and execution semantics of Pacman game in SimpleGT). Corresponding verification times are shown in the *third* column.

The first contract (*gemReachable*) we verified is 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 contain the gem, and 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.

```
procedure main();
/* inv: PacmanSurvive */
requires (∀ gs1: ref • (gs1∈find(srcHeap,pacman$GameState) ∧
read(srcHeap,gs1,pacman$GameState.STATE)==STATE.GhostMove) ⟹
  (∀ grid1: ref • grid1∈find(srcHeap,pacman$Grid) ∧
dtype(read(srcHeap,grid1,pacman$Grid.hasEnemy))<: pacman$Ghost ⟹
    ¬(dtype(read(srcHeap,grid1,pacman$Grid.hasPlayer))<: pacman$Pacman) ));
...
modifies srcHeap;
/* inv: PacmanSurvive */
ensures (∀ gs1: ref • (gs1∈find(srcHeap,pacman$GameState) ∧
read(srcHeap,gs1,pacman$GameState.STATE)==STATE.GhostMove) ⟹
  (∀ grid1: ref • grid1∈find(srcHeap,pacman$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 (p: Seq ref);
... // Boogie contract for the execution semantics of match step.
procedure apply_PlayerMoveLeft(p: Seq ref);
... // Boogie contract for the execution semantics of apply step.
```

Fig. 4: Boogie encoding to verify the correctness of the *Pacman* transformation

The second contract (*PacmanSurvive*) we verified is that there exists a path where the ghost never kills Pacman. Our verification strategy is to provide a path that witnesses 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. 2).

The third contract (*PacmanMoved*) we verified 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 [12]) for this task. Contract-only variables do not participate in the runtime execution of a program, 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 move toward any direction. 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. 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

	Boogie (LoC)	Veri. Time (s)
gemReachable	598	0.998
PacmanSurvive	587	1.747
PacmanMoved	579	0.109
Total	1764	2.854

Table 1: Performance measures for verifying transformation correctness of *Pacman*

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.

We also use our EMFTVM library in the VeriMTLr framework to encode the runtime behaviour of SimpleGT in Boogie. Consequently, we can verify that our encoding of the execution semantics of SimpleGT soundly represents its corresponding runtime behaviour using the translation validation approach. Due to space limitations, we are unable to explain the details of our verification for the sound encoding of the execution semantics of SimpleGT. We refer to our previous work for how to do this [9]. The generated Boogie programs for the Pacman game (including the soundness encoding verification and OCL transformation contracts verification) can be found in our online repository [8].

5 Related and Future Work

Model transformation verification is an active research area [1]. In this section, we will focus on GT verification. Syriani and Vangheluwe propose an input-driven simulation approach using the Discrete EVent system Specification (DEVS) formalism [15]. Bill et al. extend OCL with CTL-based temporal operators to express properties over the lifetime of a graph [5]. 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 from the same expressibility issue as any other FOL-based verifier. 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 [2]. 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 structural contracts of GT [14]. The transformation rules are given as a textual description based on a relational calculus. The formalizations 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. In addition to this, we are able to address a different challenge. That is we also verify that the execution semantics of GT encoded in Boogie faithfully 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 [13]. An earlier proposal to adapt translation validation approach in GT verification was also made by Horváth [11].

Finally, the two most widely used intermediate verification languages are Boogie, and Why3 [10]. Both languages have mature implementations and frameworks to parse, type-check, and analyse programs. We concentrate on Boogie in this paper, but all results can be carried over to Why3, or to other verification languages with similar functionality.

Our future work will concentrate on automating the compilation from SimpleGT to Boogie, and its integration into Eclipse with a user interface.

References

1. Amrani, M., Lucio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y., Cordy, J.R.: A tridimensional approach for studying the formal verification of model transformations. In: ICST. pp. 921–928 (2012)
2. Asztalos, M., Lengyel, L., Levendovszky, T.: Formal specification and analysis of functional properties of graph rewriting-based model transformation. Softw. Test., Verif. Reliab. 23(5), 405–435 (2013)
3. ATLAS Group: Specification of the ATL virtual machine. Tech. rep., Lina & INRIA Nantes (2005)
4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. pp. 364–387 (2006)
5. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: Model checking of CTL-extended OCL specifications. In: SLE. pp. 221–240 (2014)
6. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC. pp. 102–126 (2000)
7. Cheng, Z.: Formal verification of relational model transformations using an intermediate verification language. In: MODELSWARD (Doctoral Consortium) (2015)
8. Cheng, Z., Monahan, R., Power, J.F.: Online repository for VeriGT system. https://github.com/veriatl/verigt (2015)
9. Cheng, Z., Monahan, R., Power, J.F.: A sound execution semantics for ATL via translation validation. In: ICMT. p. To appear (2015)
10. Filliâtre, J.C.: One logic to use them all. In: CADE. pp. 1–20 (2013)
11. Horváth, Á.: Towards a two layered verification approach for compiled graph transformation. In: ICGT. pp. 499–501 (2008)
12. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR. pp. 348–370 (2010)
13. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL. pp. 42–54 (2006)
14. Schätz, B.: Verification of model transformations. In: GT-VMT (2010)
15. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. SoSyM 12(2), 387–414 (2013)
16. Varró, G., Varró, D., Friedl, K.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. In: GraMoT. pp. 191–205 (2006)
17. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: MoDELS. pp. 623–637 (2011)