A Sound Execution Semantics for ATL via Translation Validation

Research Paper

Zheng Cheng(✉), Rosemary Monahan, and James F. Power

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

Abstract. In this work we present a translation validation approach to encode a sound execution semantics for the ATL specification. Based on our sound encoding, the goal is to soundly verify an ATL specification against the specified OCL contracts. To demonstrate our approach, we have developed the VeriATL verification system using the Boogie2 intermediate verification language, which in turn provides access to the Z3 theorem prover. Our system automatically encodes the execution semantics of each ATL specification (as it appears in the ATL matched rules) into the intermediate verification language. Then, to ensure the soundness of the encoding, we verify that it soundly represents the runtime behaviour of its corresponding compiled implementation in terms of bytecode instructions for the ATL virtual machine. The experiments demonstrate the feasibility of our approach. They also illustrate how to automatically verify an ATL specification against specified OCL contracts.

Keywords: Model transformation verification · ATL · Automatic theorem proving · Intermediate verification language · Boogie

1 Introduction

Model-driven engineering (MDE) has been recognised as an effective way to manage the complexity of software development. Model transformation is widely acknowledged as a principal ingredient of MDE. Two main paradigms for developing model transformations are the operational and relational approaches. Operational model transformations are imperative in style, and focus on imperatively describing **how** a model transformation should progress. Relational model transformations (MTr) have a "mapping" style, and aim at producing a declarative specification that documents **what** the model transformation intends to do. Typically, a declarative specification is compiled into a low level transformation

Z. Cheng—Funded by the Doctoral Teaching scholarship, John & Pat Hume scholarship and Postgraduate Travel fund from Maynooth University.

D. Kolovos and M. Wimmer (Eds.): ICMT 2015, LNCS 9152, pp. 133–148, 2015.
DOI: 10.1007/978-3-319-21155-8_11

implementation and is executed by the underlying virtual machine. Because of its mapping-style nature, a MTr is generally easier to write and understand than an operational transformation.

The Atlas Transformation Language (ATL) is one of the most widely used MTr languages in industry and academia [9]. An ATL specification (i.e. an ATL program) is a declarative specification that documents what the ATL transformation intends to do. It is expressed in terms of a list of rules (Sect. 2). These rules describe the mappings between the source metamodel and the target metamodel, using the Object Constraint Language (OCL) for both its data types and its declarative expressions. Then, the ATL specification is compiled into an ATL Stack Machine (ASM) implementation to be executed.

Verifying the correctness of the ATL transformation means proving assumptions about the ATL specification. These assumptions can be made explicitly by transformation developers via annotations, so-called contracts. The contracts are usually expressed in OCL for its declarative and logical nature. Many approaches have been adopted to verify the correctness of an ATL transformation [5,6,8,15]. These approaches usually consist of encoding the execution semantics of an ATL specification in a formal language. Combined with a formal treatment of transformation contracts, a theorem prover can be used to verify the ATL specification against the specified contracts. The result of the verification will imply the correctness of the ATL transformation.

However, existing approaches do not verify that the encoded execution semantics of an ATL specification soundly represents the runtime behaviour provided by the ASM implementation. Therefore, an unsound encoding will yield unsound results after verification, i.e. it will lead to erroneous conclusions about the correctness of the ATL transformation (Sect. 2). In a model transformation verification survey by Rahim and Whittle, this problem is characterised as ensuring the semantics preservation relationship between a declarative specification and its operational implementation, which is an under-researched area in MDE [1].

In this work, we are specifically interested in the core component of ATL, i.e. ATL matched rules. We aim for the sound verification of the total correctness of an ATL transformation. Therefore, we compositionally verify the termination, and the soundness of our encoding of the execution semantics of each ATL matched rule in the given ATL specification (i.e. we verify that the execution semantics of each ATL matched rule soundly represents the runtime behaviour of its corresponding ASM implementation). Consequently, we are able to soundly verify the ATL specification against its specified OCL contracts, based on our sound encodings for the execution semantics of the ATL matched rules.

We have developed our VeriATL verification system in the Boogie intermediate verification language (Boogie) to demonstrate our approach (Sect. 6) [4].

Boogie. Boogie is a procedure-oriented language that is based on Hoare-logic. It provides imperative statements (such as assignment, if and while statements) to implement procedures, and supports first-order-logic contracts (i.e. pre/postconditions) to specify procedures. Boogie 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 as its underlying theorem prover. Using Boogie in verifier design has two advantages. First, Boogie encodings can be encapsulated as libraries, which are then reusable when designing verifiers for other languages. Second, Boogie acts as a bridge between the front-end model transformation language and the back-end theorem prover. The benefit here is that we can focus on generating verification tasks for the front-end language in a structural way, and then delegate the task of interacting with theorem provers to the Boogie verifier.

Thus, using Boogie enables Hoare-logic-based automatic theorem proving via an efficient theorem prover, i.e. Z3[1]. The details for performing our proposed verification tasks were far from obvious to us, and articulating them is the main contribution of this work. In particular,

- We adapt a memory model used in the verification of object-oriented programs to explain concepts within MDE. This allows the encoding of both these MDE concepts and the execution semantics of ATL matched rules in Boogie (Sect. 4).
- We use the translation validation approach to compositionally verify the soundness of our Boogie encoding for the execution semantics of an ATL matched rule (Sect. 5). The benefit is that we can automatically verify the soundness of each ATL specification/ASM implementation pair. Our translation validation approach is based on encoding a translational semantics of the ASM language in Boogie, to allow us precisely explain the runtime behaviour of ASM implementations (Sect. 5).

2 Motivating Example

We use the ER2REL transformation as our running example [5]. It transforms the Entity-Relationship (ER) metamodel (Fig. 1(a)) into the RELational (REL) metamodel (Fig. 1(b)). Both the ER schema and the relational schema have a commonly accepted semantics. Thus, it is easy to understand their metamodels.

The *ER2REL* specification is defined via a list of ATL matched rules in a mapping style (Fig. 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 passes. Each rule also has a *to* section which specifies the elements to be created in the target

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

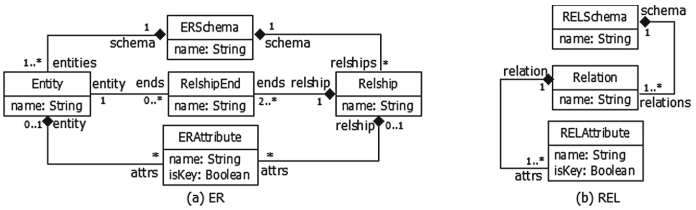

Fig. 1. Entity-Relationship and Relational metamodels

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

Fig. 2. ATL specification for ER2REL model transformation

model. The rule initialises the attribute/association of a generated target element via the binding operator (`<-`). This binding operator resolves its right hand side before assigning to the left hand side. For example, the binding *relation*`<-`*ent* in the *EA2A* rule on line 15 of Fig. 2 assigns the *Relation* element that is created for *ent* by the *R2R* rule to the *relation*.

3 Proving Transformation Correctness

In this work the correctness of an ATL transformation is specified using OCL contracts. These OCL contracts form a Hoare-triple which is used to verify the

```
1 context ERSchema inv entities_unique: -- entity names are unique in the ER schema
2   self.entities->forAll(e1,e2 | e1<>e2 implies e1.name<>e2.name)
3 ----------------------------------------------------------------
4 context RELSchema inv relations_unique: -- relation names are unique in the REL schema
5   self.relations->forall(r1,r2| r1<>r2 implies r1.name<>r2.name)
```

Fig. 3. OCL contracts for ER and REL

correctness of each ATL transformation. For example, using the OCL contracts specified in Fig. 3, we can verify whether the constraint *entities_unique* imposed on the *ER* metamodel, along with the *ER2REL* specification, guarantees that the constraint *relations_unique* holds on the *REL* metamodel.

In order to prove the correctness of the ATL transformation, we encode the OCL transformation contracts, along with the ATL transformation specification into the Boogie language. Figure 4 shows this encoding applied to the *ER2REL* transformation:

– 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 2–8), and the OCL constraints on the target metamodels are encoded as Boogie postconditions (line 10–16).

```
1  procedure main();
2  /* precondition: entity names are unique in the ER schema */
3  requires (∀ s: ref • s∈find(srcHeap,ER$ERSchema) ⟹
4   (∀ j1,j2: int • 0≤j1<j2<arrayLength(read(srcHeap,s,ERSchema.entities)) ⟹
5    read(srcHeap,s,ERSchema.entities)[j1] ≠
6    read(srcHeap,s,ERSchema.entities)[j2] ⟹
7     read(srcHeap,read(srcHeap,s,ERSchema.entities)[j1],Entity.name) ≠
8     read(srcHeap,read(srcHeap,s,ERSchema.entities)[j2],Entity.name)));
9  modifies tarHeap;
10 /* postcondition: relation names are unique in the REL schema */
11 ensures (∀ t: ref • t∈find(tarHeap,REL$RELSchema) ⟹
12  (∀ j1,j2: int • 0≤j1<j2<arrayLength(read(tarHeap,t,RELSchema.relations)) ⟹
13   read(tarHeap,t,RELSchema.relations)[j1] ≠
14   read(tarHeap,t,RELSchema.relations)[j2] ⟹
15    read(tarHeap,read(tarHeap,t,RELSchema.relations)[j1],Relation.name) ≠
16    read(tarHeap,read(tarHeap,t,RELSchema.relations)[j2],Relation.name)));
17
18 implementation main() {
19 /* Initialize Target model */
20   call init_tar_model();
21 /* instantiation phase */
22   call S2S_matchAll(); call E2R_matchAll(); call R2R_matchAll();
23   call EA2A_matchAll(); call RA2A_matchAll(); call RA2AK_matchAll();
24 /* initialisation phase */
25   call S2S_applyAll(); call E2R_applyAll(); call R2R_applyAll();
26   call EA2A_applyAll(); call RA2A_applyAll(); call RA2AK_applyAll();
27 }
```

Fig. 4. Verifying the Correctness of the *ER2REL* Transformation

– Second, the execution semantics of the ATL specification is encoded as a Boogie implementation (line 18–27). The body of this Boogie implementation is a series of procedure calls to the encoded Boogie contracts for the execution semantics of each ATL matched rule. Specifically, the execution semantics of a given matched rule involves an **instantiation** step (for matching source elements and allocating target elements) and an **initialisation** step (for initialising target elements) [3]. Each step is encoded as a Boogie contract. These Boogie contracts for ATL rules are scheduled to execute their instantiation steps before their initialisation steps, which ensures the confluence of transformation [3].
– Finally, we pair the Boogie contract that represents the specified OCL contracts, with the Boogie implementation that represents the execution semantics of the

ATL specification. 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 specification satisfies the specified OCL contracts, or trace information that indicates where the OCL contract violation is detected.

Whether the ER2REL transformation is verified for the given OCL contracts depends on our encoded Boogie contracts for the execution semantics of each ATL matched rule. Our encoding is based on existing documentation of ATL [3,9]. However, the ambiguities in the documentation increase our encoding difficulty. For example, on line 5 of the ER2REL specification (Fig. 2), the *relations* association is bound twice. The ATL documentation does not explicitly specify how to encode the execution semantics of such a case. We can encode it by either assuming that:

- The second binding *overwrites* the first binding. In this case the *relations_unique* constraint holds, since the *relations* of each *RELSchema* element will be resolved from the *entities* of the *ERSchema* element only; or
- The second binding is *composed* with the first binding. In this case the *relations_unique* constraint does not hold, since the *relations* of each *RELSchema* element will come from both the *relships* and *entities* of the *ERSchema* element, and 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.

Problem Statement. To resolve the ambiguity here, our quest in this work is to ensure our encoded execution semantics of the ATL specification soundly represents the runtime behaviour of its corresponding ASM implementation, i.e. verifying the soundness of our encoding for the execution semantics of the ATL specification. Therefore, in the next sections, we first detail our Boogie encoding for the execution semantics of each ATL matched rule (Sect. 4). Then, we report our translation validation approach to verify the soundness of our encoding (Sect. 5).

4 Encoding Metamodels, OCL and ATL Matched Rules

To begin with, we illustrate how to encode the metamodels and OCL constructs in Boogie, which will be used to encode the execution semantics of ATL matched rules.

Metamodels. Metamodelling concepts share many similarities with object oriented (OO) programming language constructs. Thus, we reuse the encoding of OO programs (specifically the memory model) to encode metamodels in Boogie.

Specifically, each classifier in the metamodel gives rise to a unique constant of type $ClassName$. Inheritance is defined via a partial order between two classifiers (multiple-inheritance is currently not supported by our encoding). Each element of a classifier is abstracted as a reference and generated as a Boogie variable of type ref. Each structural feature is mapped to a unique constant of type

$Field\ \alpha$, where α is of primitive type (i.e. *int*, *bool* and *string*) for an attribute, and is of ref type for an association. Moreover, all these constants generated for attributes or associations are extended with the corresponding classifier name to ensure their uniqueness.

The OO memory model we chose uses an updatable array *heap* to organise the relationships between model elements. The *heap* is of type $ref \times (Field\ \alpha) \rightarrow \alpha$. Thus, it maps memory locations (identified by an element of a classifier, and a structural feature) to values. A memory access expression $o.f$ is now seen as an expression $read$(*heap*,o,f). An assignment $o.f := x$ is understood as an expression $update$(*heap*,o,f,x), i.e. changing the value of heap at the position given by the element o and structural feature f to the value of x. In addition, the domain of the *heap* includes allocated as well as unallocated elements. To distinguish between these two, we add a structural feature *alloc* of type $Field\ bool$ and arrange to set it to true when an element is allocated.

OCL Constructs. We encode a subset of OCL data types supported in ATL, i.e. OclType, Primitive (*bool*, *int* and *string*), Collection (*set*, *ordered-set*, *sequence*, *bag*) and Map data types. Overall, 32 OCL operations are supported on the chosen data types. This encoding is based on a Boogie library for the theory of *set*, *sequence*, *bag* and *map* provided by the **Dafny** verification system [11]. Twenty-three Boogie functions from this library are directly reused in our encoding. One of them is modified to enhance the verification performance for sequence slicing. On top of these, we further introduce the *ordered-set* collection data type (with 3 OCL operations), and 6 OCL iterators on *sequence* and *ordered-set* data types (i.e. exists, forall, isUnique, select, collect and reject iterators). One subtlety in our encoding of OCL is how to handle the two $Undefined$ values (i.e. *null* and *invalid*). To simplify the type system, we decided to support *null* as the $Undefined$ value exclusively, and have not encountered verification problems caused by this decision.

ATL Matched Rules. According to the specification of the ATL virtual machine [3], the execution semantics of a given matched rule involves an instantiation step and an initialisation step. The execution semantics of each step is encoded as a Boogie contract.

We introduce three functions to help our encoding. The *dtype* function returns the classifier for a given reference. The $find$ function returns all the references for the given classifier allocated on the given *heap*. 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.

As an example, the instantiation step for the $S2S$ rule is shown in Fig. 5. First, it requires that the target element generated for the $ERSchema$ source element is not allocated yet (line 2–3). Then, it specifies that the instantiation step will only affect the heaps for the target model (line 4). This is because 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

```
1  procedure S2S_matchAll();
2  requires (∀ s:ref • s∈find(srcHeap,ER$ERSchema)⟹
3    getTarget({s})=null ∨ ¬read(tarHeap,getTarget({s}),alloc));
4  modifies tarHeap;
5  ensures (∀ s:ref • s∈find(srcHeap,ER$ERSchema)⟹
6    read(tarHeap,getTarget({s}),alloc)
7    ∧ getTarget({s})≠null
8    ∧ dtype(getTarget({s}))=REL$RELSchema);
9  ensures (∀ o:ref, f:Field α •
10   (o=null ∨ read(tarHeap,o,f)=read(old(tarHeap),o,f)
11   ∨ (dtype(o)=REL$RELSchema
12     ∧ f=alloc ∧ dtype(getTarget_inverse(o)[0])=ER$ERSchema)));
```

Fig. 5. The auto-generated Boogie contract for the instantiation step of the *S2S* rule

heap will not affect the state of the source heap. Next, it ensures that after the execution of the instantiation step, for each *ERSchema* element, the corresponding *RELSchema* target element is allocated (line 5–8). Finally, it ensures that nothing else is modified, except the *RELSchema* element(s) created from the *ERSchema* element by the instantiation step (line 9–11).

```
1  procedure S2S_applyAll();
2  requires (∀ s:ref • s∈find(srcHeap,ER$ERSchema)⟹
3    read(tarHeap,getTarsBySrcs({s}),alloc)
4    ∧ getTarget({s})≠null ∧ dtype(getTarget({s}))=REL$RELSchema);
5  modifies tarHeap;
6  ... // t.relations≠null ∧ t.relations.alloc
7  ... // dtype(t.relations)=class._System.array
8  // length(t.relations)=length(s.entities)+length(s.relships)
9  ensures (∀ s:ref • s∈find(srcHeap,ER$ERSchema)⟹
10   ArrayLength(read(tarHeap,getTarsBySrcs({s}),RELSchema.relations))
11   = ArrayLength(read(srcHeap,s,ERSchema.entities))
12     +ArrayLength(read(srcHeap,s,ERSchema.relships))
13 );
14 // t.relations[j] = resolve(s.entities[j])
15 ensures (∀ s:ref • s∈find(srcHeap,ER$Entity)⟹
16   (∀ j:int • 0≤j<ArrayLength(read(srcHeap,s,ERSchema.entities))⟹
17     read(tarHeap,getTarsBySrcs({s}),RELSchema.relations)[j]
18     =getTarsBySrcs({read(srcHeap,s,ERSchema.entities)[j]})));
19 // t.relations[j+len(s.entities)] = resolve(s.relships[j])
20 ensures (∀ s:ref • s∈find(srcHeap,ER$Entity)⟹
21   (∀ j:int • 0≤j<ArrayLength(read(srcHeap,s,ERSchema.relships))⟹
22     read(tarHeap,getTarsBySrcs({s}),RELSchema.relations)
23       [j+ArrayLength(read(srcHeap,s,ERSchema.entities))]
24     =getTarsBySrcs({read(srcHeap,s,ERSchema.relships)[j]})));
25 ensures (∀ o:ref,f:Field α •
26   o≠null ∧ read(old(tarHeap),o,alloc)⟹
27   (dtype(o)=REL$RELSchema ∧ f=RELSchema.relations
28     ∧ dtype(getTarget_inverse(o)[0])=ER$ERSchema)
29   ∨ (read(tarHeap,o,f)=read(old(tarHeap),o,f)));
```

Fig. 6. The auto-generated Boogie contract for the initialisation step of the *S2S* rule

The Boogie contract generated for the initialisation step of the *S2S* rule is shown in Fig. 6. First, it requires that the instantiation step of the *S2S* rule is finished (line 2–4). Then, it specifies that only the heap for the target model will be modified (line 5). Next, it ensures that the corresponding target element is fully initialised, by performing associated binding as specified in the *S2S*

rule (line 6–24). In particular, we encode consecutive bindings to the *relations* association as a composition. Finally, it ensures that nothing else is modified, except the binding performed on the created target element (line 25–29).

5 Sound Encoding for the Execution Semantics of ATL Rules

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). An important contribution of our work 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 soundly represents the runtime behaviour of its corresponding ASM operation. In this section, we first provide a translational semantics of the ASM language in Boogie, which allows the runtime behaviour of the ASM operations to be represented using Boogie implementations. Then, we illustrate our translation validation approach to verify the soundness of our Boogie encoding for the execution semantics of ATL rules.

Translational Semantics of ASM. Each ASM 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, and this is abstracted as an OCL *sequence* data type, which is represented as a list in Boogie called *Stk* in our encodings. 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 Sect. 4.

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

We provide a 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. The only resource we can find to explain the operational semantics of ASM bytecode instructions is the specification of the ATL virtual machine [3]. However, it is imprecise and leaves many issues open. 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 might have different operational semantics for different model management systems. This is because ATL aims at interacting with various model management systems which offer different interfaces for model manipulation [9].

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 rule correspondingly.

In what follows, we pick a representative ASM instruction as an example, i.e. the *SET* instruction. We first give an informal description of its operational semantics, and then explain the intuition behind its corresponding translation to Boogie. The full translational semantics of the ASM language can be accessed through our online repository given in Sect. 6.

The *SET* instruction is one of the ASM instructions for model handling (Fig. 7 (left)). 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 in Sect. 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 will be a composition of two bindings.

```
n: SET f
```

```
let o=hd(tl(Stk)),v=hd(Stk) in
  assert size(Stk)>1 ∧ o≠null ∧ read(heap,o,alloc);
  if(isCollection(f))
   {heap:=update(heap,read(heap,o,f),read(heap,o,f)∪v);}
  else {heap:=update(heap,o,f,v);}
  Stk:=tl(tl(Stk));
```

Fig. 7. *SET* instruction in ASM (left) and its translation rule in Boogie (right)

The translation rule for the *SET* instruction is shown in Fig. 7 (right). It offers no surprise in its operational semantics, except for the auxiliary function *isCollection*. The *isCollection* function (of type $Field\ \alpha \rightarrow bool$) is encoded while mapping the structural features 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.

The translational semantics of the ASM language is encapsulated as a Boogie library, which can be found in our online repository as outlined in Sect. 6.

Translation Validation of Encoding Soundness. In order to verify the soundness of our Boogie encoding for the execution semantics of each ATL matched rule, we define that the execution semantics of an ATL matched rule encoded in Boogie is sound, if,

```
1  procedure S2S_matchAll();      //Contract for instantiation step
2  ...
3  ensures (∀ s:ref • s∈find(srcHeap,ER$ERSchema)⟹
4      dtype(getTarget({s}))=REL$RELSchema);
5
6  implementation S2S_matchAll() //Implementation for matchAll operation
7  { ...
8    #ERSchemas := find(srcHeap,ER$ERSchema);
9    counter:=0;
10
11   while(counter<size(#ERSchemas))
12     invariant (∀ n:int • 0≤n<counter⟹
13       dtype(getTarget({#ERSchemas[n]}))=REL$RELSchema);
14     decreases size(#ERSchemas)-counter;
15   { ... counter:=counter+1; }
16 }
```

Fig. 8. Soundness verification of Boogie encodings for the instantiation step of $S2S$ rule

- 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 them forms a verification task, and 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 ATL matched rules is sound. Essentially, our approach is based on a translation validation technique used in compiler verification [12]. The benefit is that we do not need to verify that the encoded execution semantics of ATL specifications 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 specification/ASM implementation pair.

We demonstrate our approach on the instantiation step of the $S2S$ rule (Fig. 8). Generally, 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** that holds before and after the loop. The general loop invariant for the Boogie implementation is automatically generated. This is demonstrated on the soundness verification of Boogie encodings for the instantiation step of the $S2S$ rule as follows (Fig. 8): 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 12–13). 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).

We also use a **variant expression** to ensure that the loop terminates. A general variant expression for the Boogie implementation of a *matchAll* operation

is the size of the iterated collection minus the increasing loop counter (line 14). 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.

We can conclude that the execution semantics of an ATL specification encoded in Boogie is sound when the execution semantics of all the relevant ATL matched rules encoded in Boogie are sound.

6 Implementation

We have implemented the **VeriATL** verification system (Fig. 9) to demonstrate our approach. It accepts the source and target ECore metamodels and an ATL specification. The output is a sound execution semantics of the ATL specification encoded in the Boogie intermediate verification language, which soundly represents the runtime behaviour of its corresponding ASM implementation. As a result, the verification of the correctness of the ATL transformation that is based on our output will be sound.

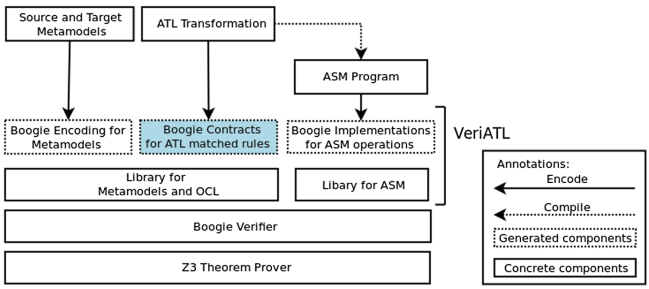

Fig. 9. Overview of our sound verification of the correctness of the ATL transformation

VeriATL automatically serialises its inputs into three kinds of models. Specifically, the KM3 API is used to serialise the ECore metamodels into the KM3 model[2]. The ATL extractor API is used to serialise the input ATL specification as an ATL model. The ATL virtual machine API is used to serialise the ASM program into an ASM model. Next, the corresponding Boogie code is automatically generated for each kind of model by a template-based model-to-text transformation using Xpand[3], i.e. the ATL model generates Boogie contracts, the KM3 model generates Boogie types and constants, and the ASM model produces Boogie implementations. Then, VeriATL sends the generated Boogie code to the Boogie verifier (version 2.2), and relies on the Z3 (version 4.3) to perform automatic theorem proving. Finally, if the Boogie verifier confirms that the execution semantics of an ATL specification encoded in Boogie is sound, then such

[2] KM3 is a domain specific language for metamodel specifications.
[3] Xpand. http://wiki.eclipse.org/Xpand/.

an encoding will be output by VeriATL. Otherwise, the trace information from the Boogie verifier, indicating where the encoding unsoundness was detected, will be output.

Evaluation. We evaluate VeriATL on the *ER2REL* transformation. Our *ER2REL* transformation is a modified version of the one originally developed by Büttner et al. [5]. The modification does not cause the ATL specification 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.

Our experiment is performed on an Intel 2.93 GHz machine with 4 GB of memory running on Windows. Verification times are recorded in seconds. Table 1 shows the performance on automatically verifying the soundness of our Boogie encoding. 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/Boogie implementation). Their corresponding verification time is shown in the *fourth* and *fifth* columns.

Table 1. Performance measures for verifying the encoding soundness of *ER2REL*

Rule name	Boogie (LoC)		Veri. time (s)		Automation
	Instantiation	Initialisation	Instantiation	Initialisation	
S2S	13/133	41/200	0.124	0.894	Auto
E2R	13/150	15/79	0.109	0.077	Auto
R2R	13/150	15/79	0.109	0.062	Auto
EA2A	17/202	33/145	0.187	0.328	Auto
RA2A	17/202	33/145	0.187	0.327	Auto
RA2AK	17/225	33/141	0.374	0.311	Auto
Total	90/1062	170/789	1.090	1.999	

We also verify our modified *ER2REL* transformation against the 4 OCL contracts that are specified by Büttner et al., and produce the same verification result. Table 2 shows the performance of our transformation correctness verification. The *second* column shows the size of the Boogie code generated for the OCL contracts. Its corresponding verification time is shown in the *third* column. In addition, we report that 2 out of 4 OCL contracts are verified semi-automatically. This is because of incompleteness issues with our approach, which we analyse in the **threat to validity** section below.

Due to space limitations, we are unable to show the whole case study. We refer to our online repository for the generated Boogie programs for verifying the correctness of *ER2REL* transformation [7].

Threat to Validity. The experiments strongly demonstrate the feasibility of our approach. However, our current approach has some limitations:

Table 2. Performance measures for verifying transformation correctness of *ER2REL*

	Boogie (LoC)	Veri. time (s)	Automation
unique_rel_schema_names	45	0.624	Auto
unique_rel_relation_names	48	1.716	Semi
unique_rel_attribute_names	48	0.608	Auto
exist_rel_relation_iskey	49	0.562	Semi
Total	190	3.510	

- First, the soundness of our approach depends on the correctness of our encodings for metamodels, OCL, ATL language and ASM bytecode. The correctness of these encodings are challenging theoretical problems that require well-defined and commonly accepted formal semantics of each. To our knowledge, none of them are currently available. When there is one, we can adapt existing techniques to reason the correctness of our encodings [2,8]. Moreover, our Boogie encodings are intuitive and available for inspection.
- Second, the completeness of our approach remains one of the major concerns. The incompleteness might be due to known limitations of SMT solvers. It may also be due to our encodings. For example, the *append* operation of *sequence* data type in our OCL library is encoded by the essential axioms to define its meaning. The auxiliary axioms such as "any sequence appended with an empty sequence is the original sequence" are not in our encoding. We think it is better to present the missing auxiliary axioms as lemmas and introduced on demand to make the verification task smaller. Moreover, presenting only the essential axioms is a strategy that helps manual inspection and reduces the possibility of inconsistent axioms.
- Third, our approach only covers the ATL matched rules in this work. Other constructs, such as lazy rules and imperative features (e.g. *resolveTemp* operation), are not considered. We would like to include them in the near future. For example, we are currently considering ATL lazy rules, which are called from the other rules. The lazy rules are not as frequently used as the matched rules, but are the main source of transformation non-termination.
- Last, because of the underlying SMT solver, the expressiveness of our approach is based on first order predicate logic with equality. To ensure this expressiveness power is useful in practice of MTr verification, we need to experiment with more ATL transformations that have OCL contracts specified.

7 Related Work

There is a large body of work on the topic of ensuring model transformation correctness [1]. In this section, we focus on the works that verify the correctness of MTr by applying formal methods.

Troya and Vallecillo provide an operational semantics for ATL based on rewriting logic, and use the Maude system for the simulation and reachability

analysis of ATL specifications [15]. Lúcio et al. develop an off-the-shelf model checker that is tied to the DSLTrans language. Their model checker allows the user to check the syntactic correctness (encoded in algebra) of the generated target models [13]. These approaches are bounded, which means that 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 attributes). Bounded approaches are usually automatic, but no conclusion can be drawn outside the search space.

Calegari et al. use the Coq proof assistant to interactively verify that an ATL specification is able to produce target models that satisfy the given contracts [6]. Inspired by the proof-as-program methodology, further research develops the concept of proof-as-model-transformation methodology [10,14]. At its simplest, the idea is to present the model transformation specification and contract as a theorem. Then, a model transformation implementation can be extracted from its proof. These approaches are unbounded. Therefore, they are preferable when the user requires that contracts hold for the MTr specification over an infinite domain. However, unbounded approaches tend to require guidance from the user.

The situation can be ameliorated by a novel usage of SMT-solvers. The built-in background theories of SMT solvers give enhanced expressiveness to handle constraints over an infinite domain. For example, Büttner et al. translate a declarative subset of the ATL and OCL contracts (for semantic correctness) directly into first-order-logic formulas [5]. The formulas represent the execution semantics of the ATL 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 specified OCL contracts. However, their approach lacks an intermediate form to bridge between the ATL and the back-end SMT-solver. This compromises the reusability and modularity of the verifier. In our work, we extend existing Boogie libraries for our metamodel and OCL encodings. We also develop a Boogie library that gives a translational semantics to the ASM language. Each Boogie library is designed modularly, and is made available for public reuse of verifier design (especially for model transformation languages).

Finally, all the approaches we have just described rely on encoding the execution semantics of the model transformation specification. We address a different challenge to verify that the execution semantics of an ATL matched rule encoded in Boogie soundly represents the runtime behaviour of its corresponding ASM implementation, which makes our approach complementary to the existing approaches. We developed our approach in Boogie. The Why3[4] intermediate verification language would also be suitable to implement our approach.

8 Conclusion

In this work, we have encoded a sound execution semantics for ATL specifications, and developed the VeriATL verification system for this task. It is implemented in Boogie which allows Hoare-logic-based automatic theorem proving

[4] Why3. http://why3.lri.fr/.

via the Z3 theorem prover. We adapt the memory model used in the verification of object-oriented programs to explain the concepts within MDE. We explain precisely the runtime behaviour of ASM implementations by encoding a translational semantics of the ASM language in Boogie. We also articulate 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 ATL specification against its specified OCL contracts, based on our sound encodings for the execution semantics of the ATL matched rules.

References

1. Ab.Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Soft. Syst. Modeling (2015) (to appear)
2. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd edn. Springer, Berlin (2009)
3. ATLAS Group: Specification of the ATL virtual machine. Technical report, Lina & INRIA Nantes (2005)
4. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
5. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 432–448. Springer, Heidelberg (2012)
6. Calegari, D., Luna, C., Szasz, N., Tasistro, Á.: A type-theoretic framework for certified model transformations. In: Davies, J. (ed.) SBMF 2010. LNCS, vol. 6527, pp. 112–127. Springer, Heidelberg (2011)
7. Cheng, Z., Monahan, R., Power, J.F.: Online repository for VeriATL system (2013). https://github.com/veriatl/veriatl
8. Combemale, B., Crégut, X., Garoche, P., Thirioux, X.: Essay on semantics definition in MDE - an instrumented approach for model verification. J. Softw. **4**(9), 943–958 (2009)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
10. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects Comput. **27**(1), 193–235 (2015)
11. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
12. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. SIGPLAN Not. **41**(1), 42–54 (2006)
13. Lúcio, L., Barroca, B., Amaral, V.: A technique for automatic validation of model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 136–150. Springer, Heidelberg (2010)
14. Poernomo, I.H.: Proofs-as-model-transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 214–228. Springer, Heidelberg (2008)
15. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. J. Object Technol. **10**(5), 1–29 (2011)