Softw Syst Model (2018) 17:1197–1225
https://doi.org/10.1007/s10270-016-0553-x

REGULAR PAPER

Formalised EMFTVM bytecode language for sound verification of model transformations

Zheng Cheng[1] · **Rosemary Monahan**[1] · **James F. Power**[1]

Received: 9 December 2015 / Revised: 17 April 2016 / Accepted: 21 July 2016 / Published online: 16 August 2016

Abstract Model-driven engineering is an effective approach for addressing the full life cycle of software development. Model transformation is widely acknowledged as one of its central ingredients. With the increasing complexity of model transformations, it is urgent to develop verification tools that prevent incorrect transformations from generating faulty models. However, the development of sound verification tools is a non-trivial task, due to unimplementable or erroneous execution semantics encoded for the target model transformation language. In this work, we develop a formalisation for the EMFTVM bytecode language by using the Boogie intermediate verification language. It ensures the model transformation language has an implementable execution semantics by reliably prototyping the implementation of the model transformation language. It also ensures the absence of erroneous execution semantics encoded for the target model transformation language by using a translation validation approach.

Keywords MDE · EMFTVM · Boogie · Model transformation verification · Intermediate verification language

Communicated by Prof. Alfonso Pierantonio.

✉ Zheng Cheng
zcheng@cs.nuim.ie

Rosemary Monahan
rosemary@cs.nuim.ie

James F. Power
jpower@cs.nuim.ie

[1] Department of Computer Science, Maynooth University, Maynooth, Co. Kildare, Ireland

1 Introduction

Model-driven engineering (MDE) is an effective approach for addressing the full life cycle of software development [12] and can be seen as an evolution from earlier object management architecture or component-oriented technology. MDE focuses on accurately modelling the problem rather than programming it, which allows the problem to be well comprehended before generating an implementation. In addition, MDE unifies some of the best practices in software architecture, including modelling, meta-data management and model transformation (MT) technologies. Thus, it allows a user to model once and to target multiple technology implementations by using precise MTs.

MT is widely acknowledged as one of the central ingredients of MDE. A MT allows the automatic generation of a target model from a source model, according to a transformation specification [41]. Three main paradigms for developing MTs are the operational, relational and graph-based approaches [24].

With the complexity of MTs increasing to handle the needs of industries in areas such as automotive [58], medical data processing [71] and aviation [10], it is urgent to develop techniques and tools that prevent incorrect MTs from generating faulty models. The effects of such faulty models could be unpredictably propagated into subsequent MDE steps, e.g. code generation, to produce further errors. The correctness of a MT is typically defined by transformation developers using contracts. The contracts express the assumptions about those circumstances when the MT is considered to be correct. In MDE, contracts are usually expressed in OCL due to its declarative and logical nature [55].

Dedicated verification tools can be developed to reason about the correctness of a MT. Typically, such verifiers require both an encoding of the execution semantics of the

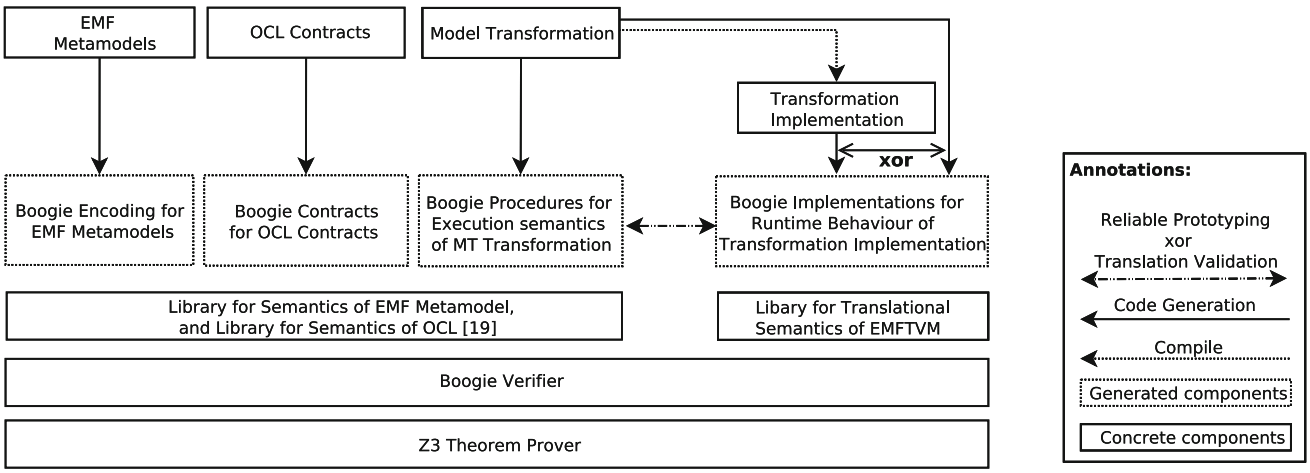

Fig. 1 Overview of our formalisation of EMFTVM bytecode language

MT language and an encoding of the transformation contracts, in a reasoning engine of a chosen logic, e.g. Hoare logic [34]. Thus, the results obtained from using this reasoning engine will imply the correctness of the MT.

Generally, when designing the verifier for a MT language, the verifier architects face two issues:

- First, if the MT language does not have an implementation yet, can the architects ensure that the encoded execution semantics of the transformation language is implementable?
- Second, if the MT language already has an implementation, can the architects ensure that the encoded execution semantics correctly represent the runtime behaviour of the underlying implementation?

If the answer to both of these questions is no, the soundness of the MT verification is compromised, i.e. the verification confirms the correctness of a MT, but the transformation generates faulty models at runtime, because:

- An unimplementable execution semantics of a MT language introduces a "miracle" to program verification. For example, assume that a MT language L contains an operator OP that operates on integers, which has an unimplementable execution semantics such as $OP(x) = OP(x) + 1$. Using the execution semantics of the OP operator in the verification would trivially render any MT that is written in L as verified [26,27,44].
- An execution semantics of a MT language that is incorrect with respect to the runtime behaviour of the underlying implementation could lead to erroneous conclusions about the correctness of the MT (see Sect. 2.2 for an example).

As visualised in Fig. 1, our solution to these issues is to provide a translational semantics for the bytecode of the EMF transformation virtual machine (EMFTVM) (Sect. 3). Specifically, our formalisation of the EMFTVM bytecode language is encoded in the Boogie intermediate verification language (IVL) [6]. Our solution contributes in two respects:

- First, it provides a formal documentation of the EMFTVM bytecode language. Thus, if a MT language has no implementation yet, our formalisation assists the verifier architects in designing its implementation correctly (Sect. 4).
- Second, our formalisation of the EMFTVM bytecode language provides an interface to Boogie. It allows the verifier designer to represent the runtime behaviour of each implementation (if there is one) into Boogie. When the execution semantics of the corresponding MT has also been represented in Boogie (facilitated by our formalisation of EMF metamodels and OCL encapsulated in Boogie [19]), soundness between the two can be verified in the state-of-the-art SMT solver Z3, via Boogie. This verification is based on each pair of MTs and their implementations. Our ideas are borrowed from the translation validation technique used in compiler verification, i.e. rather than proving that the compiler always produces target code which correctly implements the source code, each individual translation is followed by a validation phase which verifies that the target code produced on this run correctly implements the given source program [52].

  Therefore, the benefit of our proposal is that, instead of verifying that the MT is always sound with respect to its underlying implementation (which is difficult to automate), we automatically verify the soundness of each pair of source and target representations (Sect. 5).

Paper organisation Section 2 introduces the background knowledge that is required to understand our formalisation of the EMFTVM bytecode language. Section 3 illustrates our formalisation of the EMFTVM bytecode language in depth. Two major usages of this formalisation, reliable prototyping (Sect. 4) and translation validation (Sect. 5), are then presented. The feasibility of these two usages of our formalisation is shown by implementation and then evaluated in Sect. 6. Finally, Sect. 7 compares our work with related research, and Sect. 8 presents our conclusions and proposed future work. The appendix presents the full formalisation for the EMFTVM bytecode language.

2 Background

Before diving into the details of our formalisation of the EMFTVM bytecode language, we give a brief introduction to the EMFTVM bytecode language (Sect. 2.1). Then, we motivate the demand of our formalisation of the EMFTVM bytecode language by giving an concrete example of verifying contracts on MTs (Sect. 2.2). Finally, we provide an overview of the Boogie intermediate verification language, which is used to encode our formalisation (Sect. 2.3).

2.1 The EMFTVM bytecode language

EMFTVM is a stack-based virtual machine, which aims at providing a common execution semantics for the implementation of rule-based MT languages [69]. It is based on the Eclipse Modelling Framework (EMF), a framework which provides several core services that benefit the implementation of MT languages (e.g. facilities that enable viewing, navigating, editing and persisting of the models) [59]. Moreover, EMFTVM uses the EMFTVM bytecode language to implement MTs. Existing MT languages that target the EMFTVM include ATL, SimpleGT and EMFMigrate [68,69].

The EMFTVM bytecode language contains 50 bytecode instructions. Apart from the general-purpose instructions for stack handling and control flow, an important feature of the EMFTVM language is the model-handling-specific instructions that are dedicated to EMF model manipulation. For example, while relational MT languages typically use the *SET/GET* instructions of the EMFTVM bytecode language, graph-based MT languages usually use the *ADD*, *REMOVE* and *DELETE* instructions to manipulate model elements. In addition, when the model elements are ordered collections, an *INSERT* instruction is used to insert a structural feature's value at a specific index.

The EMFTVM bytecode language organises the EMFTVM instructions into code blocks. Each block contains the executable instructions, local variables and a local stack to communicate values.

Therefore, the EMFTVM bytecode language provides a modular and systematic way to execute each MT rule in stages (e.g. matching, applying), by building a set of code blocks.

2.2 Verifying contracts on model transformations

In MDE, there is an urgent need to develop verification tools which will allow users to verify the MT they developed against specified contracts. These verifiers will ensure the correctness of the MT and control its complexity.

In this work, our verifier designs are based on Hoare logic. Thus, verifying that a MT is correct with respect to the given set of contracts can be represented as a classic Hoare triple, i.e. assuming the contracts imposed on the source metamodel (precondition) hold, the safe execution of a MT should guarantee that the contracts are fulfilled on the generated target metamodel (postcondition). In MDE, contracts are usually expressed in OCL due to its declarative and logical nature.

Example 2.1 In this section, we focus on a trace-based MT language called ATL [39], to demonstrate how to specify OCL contracts on MTs. In particular, we use the ER2REL transformation in ATL as our running example [15]. It transforms models conforming to the entity–relationship (ER) metamodel (Fig. 2a) into models conforming to the RELational (REL) metamodel (Fig. 2b). Both the ER schema and

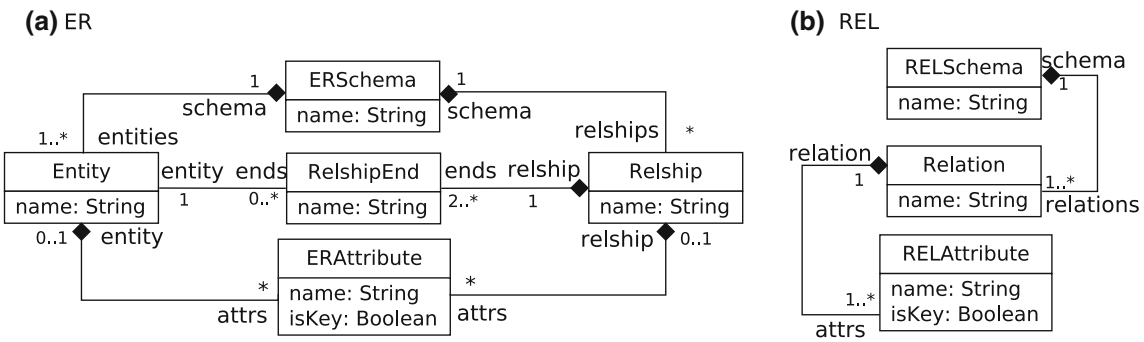

Fig. 2 Example: an entity–relationship metamodel (*left*) and a relational metamodel (*right*)

```
1  module ER2REL; create OUT : REL from IN : ER;
2
3  rule S2S {
4    from s: ER!ERSchema
5    to t: REL!RELSchema (name <- s.name, relations <- s.entities, relations <- s.relships} )}
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 ...
```

Fig. 3 Example: excerpt of an ATL transformation for *ER2REL* model transformation

the relational schema have a commonly accepted semantics. Thus, it is easy to understand their metamodels.

A snippet of the *ER2REL* transformation in ATL is shown in Fig. 3. It is defined in a mapping style via a list of ATL matched rules. 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 are similar to *EA2A* and 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 only applicable if the guard evaluates to true. Each rule also has a *to* section which specifies the elements to be created in the target model. The rule initialises the attribute/association of a generated target element via the binding operator (<-).

An important aspect for the execution semantics of ATL is the use of an implicit *resolving* algorithm during the initialisation of the target element. This algorithm is responsible for resolving the right-hand side of the binding operator before assigning to the left- hand side.

For example, in the binding *relation* <- *ent* of the *EA2A* rule (line 15 of Fig. 3), the resolving algorithm ensures that *ent* is resolved to a *Relation* element that is created by the *R2R* rule, and the resolved result will be assigned to the association *relation*.

In trace-based MT languages, the key to resolving the generated target element(s) from the given source element(s) is to use a data structure called trace. Each trace is created to link the matched source element with the created target element on each successful rule application [5].

From our previous experience of verifier development for ATL [19], we observe that there are many options for encoding the execution semantics of the resolving algorithm. This is due to the open issues documented by the ATL language specification [5]. To ensure our proposed execution semantics for the resolving algorithm is implementable, we provide a systematic approach to reliably prototyping the resolving algorithm in the EMFTVM bytecode language (Sect. 4). Our formalisation for the EMFTVM bytecode language is the foundation for such a systematic approach.

Verifying the correctness of an ER2REL transformation requires the use of the OCL contract. For example, as the OCL contracts specified in Fig. 4, the precondition *unique_er_relship_names* (imposed on the *ER* metamodel) specifies that all instances of *ERSchema* have unique names for its *relships*. The goal is to verify that the *ER2REL* transformation guarantees the postcondition *unique_rel_relation_names* (i.e. all instances of *RELSchema* have unique names for its *relations*) holds on the *REL* metamodel.

The *unique_rel_relation_names* constraint does not hold when the ATL transformation is compiled into an ASM implementation [19]. This is because in this case:

- the multiplicity of the *relations* association has an upper bound that is greater than 1 (Fig. 2).
- the *relations* association is bound twice (line *5* of the ER2REL transformation in Fig. 3). The execution semantics of the ATL transformation enforces the composition of the first and second bindings.[1]

As a result, the *relations* in each *RELSchema* element will be transformed from both the *entities* and *relships* of the *ERSchema* element. It is not guaranteed that the names of

[1] Notice that when the ATL transformation is compiled into an EMFTVM implementation, the second binding always overwrites the first binding (Sect. 6.3).

```
1 -- relship names are unique in the ER schema
2 context ER!ERSchema inv unique_er_relship_names:
3   ER!ERSchema.allInstances()->forAll(s |
4     s.relships->forAll(r1,r2 | r1<>r2 implies r1.name<>r2.name))
5 ------------------------------------------------------------
6 -- relation names are unique in RELSchema
7 context REL!RELSchema inv unique_rel_relation_names:
8   REL!RELSchema.allInstances()->forAll(s |
9     s.relations->forAll(r1,r2| r1<>r2 implies r1.name<>r2.name))
```

Fig. 4 Example: contracts for ER and REL metamodels in OCL

```
1  procedure McCarthy91(n: int) returns (r: int);
2    ensures 100 < n ⟹ r = n − 10;
3    ensures n ≤ 100 ⟹ r = 91;
4
5  implementation McCarthy91(n: int) returns (r: int)
6  { if (100 < n) {
7      r := n − 10;}
8    else {
9      call r := McCarthy91(n + 11);
10     call r := McCarthy91(r);}
11 }
```

Fig. 5 Example: McCarthy-91 function in Boogie

the *relships* are unique in each *ERSchema* element, nor that the names of *entities* and *relships* in each *ERSchema* element are different. Thus, the verification implies that the ER2REL transformation is incorrect with respect to the given OCL contracts.

We now consider what happens if a different execution semantics for the consecutive bindings to the *relations* association is chosen. For example, if a semantics where the second binding *overwrites* the first binding, the *unique_er_relship_names* constraint holds, since the *relations* of each *RELSchema* element will be resolved from the *entities* of the *ERSchema* element only. However, this verification will be unsound for the ATL transformation that compiles into the ASM implementation, because an incorrect execution semantics of ATL was used.

How can the correct execution semantics of a MT language be distinguished from an incorrect one? In our opinion, it relies on being able to (a) represent the runtime behaviour of the underlying implementation of the MT language and (b) verify it against the encoded execution semantics of the same MT language. This further motivates the need to formalise the EMFTVM bytecode language and to interface the runtime behaviour of the target MT language with Boogie.

2.3 Boogie intermediate verification language

Our formalisation of the EMFTVM bytecode language is encoded in the Boogie intermediate verification language (IVL). Boogie is a procedure-oriented IVL based on Hoare logic [6]. Each Boogie program consists of declarations for types, functions, constants, axioms, expressions, variables, procedures, or procedure implementations.

Imperative statements (such as assignment, if and while statements) are provided by Boogie to structure the procedure implementations. First-order logic (FOL) contracts (i.e. pre-/postconditions expressed by Boogie expressions) are supported and used for the specification of procedures. A comprehensive manual has been presented by Leino to give a full semantics of the Boogie IVL [47].

A Boogie program is verified if its procedure implementations satisfy their corresponding contracts. This verification is performed by the Boogie verifier which uses the Z3 theorem prover at its back end [28]. If the program is not verified, the verifier will represent the result from Z3 as program traces, to help locate the error in the Boogie program.

Example 2.2 To demonstrate what a Boogie program looks like, we show the Boogie encoding of the McCarthy 91 function in Fig. 5 [50]:

- First, the signature of the Boogie procedure specifies that the McCarthy-91 function takes one input *n* and one output *r*, both of type *int* (line 1).
- Then, the postconditions that establish the relationship between the input and output are specified by two *ensures* clauses (lines 2–3). That is, if the input is greater than 100, return the input minus 10 as the output; otherwise, always return 91.
- The procedure implementation uses a Boogie *if* statement to form a case distinction according to the input value (lines 5–11). That is, if the input value is greater than 100, the output is assigned the input minus 10. Otherwise, two recursive calls are invoked sequentially to compute the output.

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

Using Boogie in verifier design has two advantages. First, the formalisations can be encapsulated in Boogie as libraries and then reused in different verifier designs. For example, in our previous work [19], we have encoded the formalisation of EMF metamodels (based on the Burstall–Bornat memory model, see Sect. 3.1 for more detail) and a subset of OCL (i.e. datatypes such as OCLAny, OCLType, Primitive and collections, as well as corresponding operations defined on these datatypes, e.g. iterators) as Boogie libraries. In our experience, these libraries provide a foundation for the common compilation process from the EMF metamodels/OCL expressions to Boogie and thus greatly reduce the complexity of encoding the execution semantics of the target MT languages. This motivates the usage of Boogie for program verification in an integral way, taking into account the modularity and reusability of the verifier design.

Second, existing verifier designs have already established the usefulness of an IVL for decomposing the complex task of generating verification tasks for general high-level programming languages into two steps [6,31,46]: a transformation from the program and its contracts into the program written in an IVL, and then a transformation from this IVL program into verification tasks. Thus, the IVL bridges the front-end high-level programming language with the back-end theorem prover. The benefit is that users can focus on generating contracts that prescribe what correctness means for the front-end language in a structural way and can delegate the task of interacting with theorem provers to the IVL.

At the time of this paper being written, translations into Boogie exist for several languages, including C# [7], C [25], Dafny [46], Java [43] and Eiffel [64]. These show its applicability to a range of different programming styles.

3 Formalisation of EMFTVM bytecode language

In this section, we introduce the translational semantics of the EMFTVM language via a list of translation rules. Each translation rule encodes the operational semantics of an EMFTVM instruction in Boogie.

3.1 Basics of our formalisation

The only resource available regarding the EMFTVM bytecode instructions is the bytecode format for the EMFTVM virtual machine.[2] However, this documentation is imprecise and leaves many issues open (e.g. whether adding a structural feature twice on the same model element causes a runtime exception). This raises the question of how a correct translation rule, especially for each model handling instruction, should be encoded in Boogie.

Our strategy is to check the EMFTVM source code for the operational semantics of each EMFTVM instruction and then design the translation rule correspondingly.

Data structures of EMFTVM An EMFTVM implementation contains a list of EMFTVM code blocks. Each block has a list of local variables, which are encoded as local variables in Boogie. An operand stack is used by each EMFTVM block to communicate values for local computations. This is abstracted as a sequence in Boogie (called *stk* in our encoding). Source and target elements are globally accessible by every EMFTVM block, and they are managed by the disjoint source and target maps. Both maps are called *heap* in our encoding. Their encoding is based on the Burstall–Bornat memory model that is commonly used to represent the runtime heap in object-oriented verifier design [13]. This approach uses an updatable map to organise the relationships between runtime elements of classifiers, which allows the mapping of memory locations (identified by an element of a classifier and a structural feature) to values. Such a map is defined using the following Boogie type:

```
type ref;
const unique null: ref;
type Field α;
type HeapType = <α>[ref, Field α] α;
```

Boogie types include built-in types such as *bool*, *int* and *map* type (delimited by the brackets []). User-defined types can be introduced with type constructors using the keyword *type*. Here:

- *ref* is a nullary type constructor for runtime objects,
- *null* is a Boogie constant of type *ref*,
- $Field\ \alpha$ is a type constructor with one type argument to type the fields of each runtime object (e.g. a field *age* can be typed as $Field\ int$ in Boogie to indicate that it is a field of integer),
- The *HeapType* is a type synonym which abbreviates the map type that is defined on its right-hand side. Such a two-dimensional map type is defined in a polymorphic manner (i.e. parametrised by the bound type identifier α in the angle brackets $\langle\rangle$). It is specified to be the mapping from memory locations (identified by a runtime object and a field) to values of type α.

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

[2] EMFTVM bytecode format. https://wiki.eclipse.org/ATL/EMFTVM.

The domain of the *heap* includes both allocated and unallocated elements. To distinguish between these elements, we use a structural feature *alloc* of type *Field bool* which is set to true when an element is allocated. To ensure safe memory access, certain operations should only operate on the allocated elements (i.e. when *read(heap, o, alloc)* returns true).

There are several advantages to adopting the Burstall–Bornat memory model:

- it organises runtime model elements in a single updatable map, which can be flexibly passed as an argument where it is needed (e.g. as in the *read* and *update* function) [47].
- it allows quantification over fields [47] which is convenient when expressing the frame problem (Sect. 4.3).
- it enhances verifier interoperability as verifiers that are built using the same IVL and share the same memory model form a verification ecosystem, allowing information to seamlessly flow between them [21].

The full translational semantics of the EMFTVM language is given in Tables 4, 5 and 6 in the appendix, classified by the category that each EMFTVM instruction resides in. In what follows, we pick a representative EMFTVM instruction from each category and explain the intuition behind its corresponding translation rule.

3.2 Stack handling instructions

The *STORE* instruction is one of stack handling instructions. It has one operand which is a local variable that the instruction operates on. The stack is expected to be non-empty for the instruction to succeed, since it assigns the top of the stack to its operand. After the assignment, the top of the stack is then removed. Such an operational semantics for the *STORE* instruction is encoded by its corresponding translation rule in Boogie as shown in Fig. 6. In our Boogie encoding, to make sure the operand of the *STORE* instruction is declared before use, we generate a Boogie variable (denoted by ⟦x⟧, and in the same scope of the encoded *STORE* instruction) for each local variable of an ASM operation with unique name and equivalent type. In addition, we use the assert statement in Boogie to prescribe a check that the current operand stack is non-empty before executing the *STORE* instruction.

STORE x

```
assert size(stk) > 0;
⟦x⟧ := hd(stk);
stk := tl(stk);
```

Fig. 6 Formalising one of the stack handling instructions: The *STORE* instruction in EMFTVM (*left*) and its translation rule in Boogie (*right*)

IF n

```
var cond#: bool;
assert size(stk) > 0;
cond# := hd(stk);
stk := tl(stk);
if (cond#) goto ⟦n⟧;
```

Fig. 7 Formalising one of the control flow instructions: The *IF* instruction in EMFTVM (*left*) and its translation rule in Boogie (*right*)

3.3 Control flow instructions

The conditional instruction *IF* is a control instruction, which formalises a case distinction based on the boolean value that is popped of the operand stack (Fig. 7). If the popped value is true, the ASM operation continues at the instruction identified by the operand of the *IF* instruction. Otherwise, the *IF* instruction reduces to no operation. The translation rule presented here encodes this operational semantics.

In our Boogie encoding, to make sure the offset of the *IF* instruction is valid in its corresponding translation rule, we insert a fresh Boogie label, denoted by ⟦n⟧, at the program point which corresponds to the offset *n* of the *IF* instruction. Furthermore, a new Boogie variable is introduced for each *IF* instruction to hold the boolean value that is popped of the operand stack. The superscript # that is attached to the introduced Boogie variable denotes the line number of the translated *IF* instruction in an EMFTVM code block. This is to avoid name collision among introduced Boogie variables.

3.4 Model handling instructions

The *ADD* instruction is a model handling EMFTVM instruction. It has one parameter which is the structural feature to be operated upon (Fig. 8). Before executing the *ADD* instruction, the top two elements on the operand stack are a model element *o* and the value *v* (to add to *o*).

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

The translation rule for the *ADD* instruction offers no surprise based on its operational semantics, except for four points. First, a new Boogie type, *setTable*, is introduced:

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

Fig. 8 Formalising one of the model handling instructions: The *ADD* instruction in EMFTVM (*left*) and its translation rule in Boogie (*right*)

ADD 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
    { assert ¬isset(acc, o, [[f]]);
      heap := update(heap, o, [[f]], v);
      acc := set(acc, o, [[f]], true); }
  stk := tl(tl(stk));
```

This new type prohibits runtime exceptions caused by certain operations on the structural features. Such an exception could be caused by adding the same attribute twice for a model element. Thus, checking whether *o.f* is set or not becomes an expression *isset(acc, o, f)*. Marking *o.f* as set uses the expression *set(acc, o, f, true)*, and marking it as not set uses the expression *set(acc, o, f, false)*.

Second, since we use different *heaps* to represent the source and target models, the *heap* that the *ADD* instruction operates on is determined by the data type of the second from top element of the operand stack. This is accomplished by our Boogie code generator that translates the input EMFTVM code blocks to their corresponding Boogie code.

Third, ⟦f⟧denotes the corresponding Boogie constant of the parameter of the *ADD* instruction. It is of type *Field* α. For example, the *name* of *ERSchema* in Fig. 2 can be typed as *Field String* in our Boogie encoding.

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

Finally, the full translational semantics of the EMFTVM language is encapsulated as a Boogie library. We demonstrate two major usages of this library in the sections that follow. These are reliable prototyping (RP, Sect. 4) and translation validation (TV, Sect. 5).

4 Reliable prototyping for model transformation languages

One major usage of our EMFTVM bytecode language formalisation is reliably prototyping the implementations of MT languages. This is motivated by scenarios such as the target MT language not having an implementation but having a well-defined execution semantics. Thus, by ensuring its execution semantics is implementable in the EMFTVM bytecode language, we can be confident that the correctness of the MTs can be soundly verified, using this execution semantics.

Reliable prototyping of the implementation for a target MT language can be achieved by the following steps:

(RP1) Proposing the execution semantics of the target MT language and encoding it in Boogie.
(RP2) Understanding our formalised EMFTVM bytecode language and constructing a candidate implementation of the MT language using Boogie.
(RP3) Verifying the correctness of the prototype by showing that the proposed execution semantics is verified by the candidate implementation of the MT language, with the possibility of verifying its termination.

The execution semantics of recursive implementations is one of the main sources of unsoundness in verification [27]. Thus, in what follows, we show how to prototype a recursive implementation, namely the resolving algorithm of ATL, following steps RP1, RP2 and RP3 listed above.

4.1 RP1: execution semantics of resolving algorithm

In Sect. 2.2, we have introduced the resolving algorithm of ATL. In this section, we propose its execution semantics as follows (assume that the binding is expressed with the form $lhs <- resolve(y)$):

(S1) If *y* does not have a corresponding trace that stores *y* as its only source element, then *y* is returned.
(S2) If *y* does have a corresponding trace that stores *y* as its only source element, then the corresponding target element stored by the trace is returned.[3]
(S3) If *y* is of a collection type, then all of the elements in *y* are resolved individually, and the resolved results are put together into a pre-allocated collection *col*, and *col* is returned.

The proposed execution semantics and the language specification of ATL agree on the behaviour of resolving algorithm when the input has a corresponding trace that stores itself as the only source element of the trace (S2) and when the input value is of a collection type (S3) [5]. In addition to that,

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

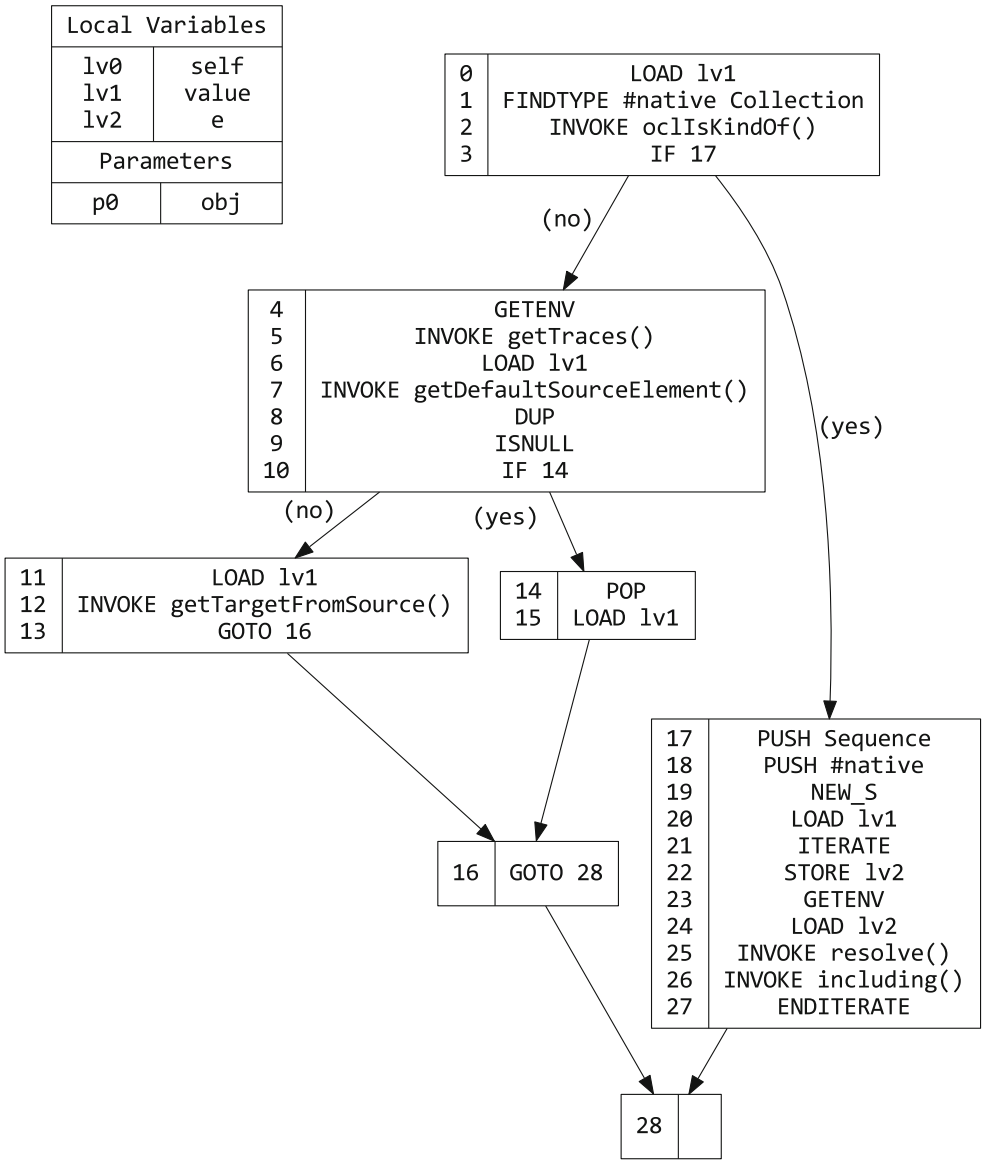

Fig. 9 Example: an implementation of the resolving algorithm in EMFTVM bytecode language (Some of its Boogie encoding is shown in Fig. 10)

we propose execution semantics *S1* to cover three boundary cases where the input does not have a corresponding trace that stores itself as the only source element of the trace, i.e. when:

- there is no corresponding ATL rule to match the input element, or
- the input is one of the generated target elements, or
- where no trace is generated for primitive types.

The proposed execution semantics *S1* is of practical use, since it is implied by the transformation implementation of ATL (e.g. ASM [38], EMFTVM [70]). However, we could not find any document that explicitly illustrates this execution semantics.

The formalised execution semantics of the resolving algorithm in Boogie follows the same structure, as shown in Sect. 4.3.

4.2 RP2: implementation of resolving algorithm

A candidate implementation of the resolving algorithm in EMFTVM is shown in Fig. 9. The implementation is based on the algorithm originally developed by Jouault et al. [39]. It accepts one model element as its parameter and declares three local variables to interact with the EMFTVM instructions for stack handling. Specifically, the local variable *self* is referenced to the model element that invokes the resolving algorithm, *value* is referenced to the input element, and *e* is a temporary variable to store the intermediate resolving result.

The resolving algorithm first forms a case distinction according to the multiplicity of the input (lines 0–4). If it is not a collection datatype (e.g. Set, Sequence, Bag or OrderedSet), then the implementation checks whether any trace holds the input value as its source (lines 5–11). If not, the input value is directly returned. Otherwise, the target value held by the trace is returned. If the input value is a collection datatype, a temporary sequence is allocated (lines 18–20). Then, each element in the input collection is resolved iteratively, and the resolved result is stored in the allocated sequence (lines 21–28). After the iteration, such a sequence holds the resolved results of the input collection and this sequence is returned as the final result of the resolving algorithm.

The formalisation of the implementation for the resolving algorithm in Boogie draws on our Boogie library for the EMFTVM bytecode language (Sect. 3) and follows the same structure as illustrated in this section. We demonstrate this in Sect. 4.3.

4.3 RP3: verification of the resolving algorithm

To verify the correctness of the resolving algorithm, we pair its execution semantics in Boogie with its corresponding formalised EMFTVM bytecode implementation. We demonstrate part of the Boogie code for this task in Fig. 10. Specifically, in the excerpt shown here, lines 16–42 of Fig. 10 correspond to lines 21–27 of Fig. 9. The full Boogie solution is 217 lines long and can be found in our online repository [20].

Lines 3–14 accommodate the execution semantics of the resolving algorithm. For example, lines 5–12 show that when the input is a collection, its elements are resolved individually, and the resolved results are put together in a temporarily allocated collection to be returned (S3).

Moreover, there are two points that should be emphasised in our formalised execution semantics of the resolving algorithm in Boogie:

- We introduce two functions to assist in the encoding. The *getTarget* function returns the corresponding target element generated for a sequence of source elements. Its inverse function *getTarget_inverse* returns the sequence of source elements used to generate the given target element.
- Our Boogie encoding addresses the **frame problem** (e.g. lines 13–14). That is, a contract of a Boogie procedure must not only specify how it affects the transformation state, but must also manifest what memory locations it will definitely not modify. The Burstall–Bornat memory model (Sect. 3.1) helps us to deal with the frame problem. First, it allows us to quantify over all the attributes/associations and specify the ones that are not affected by a binding operation. Second, we use separate *heaps* to differentiate runtime model elements (e.g. source/-trace/target/temporary models) and axiomatise them to be disjoint (an element that is allocated on one heap is not allocated on another heap). This ensures, for example, a modification made on the target heap will not affect the state of the source heap.

The formalised EMFTVM implementation of the resolving algorithm resides in lines 16–42 of Fig. 10. Some explanation is in order. First, a Boogie implementation that contains loops is difficult to verify because the users cannot generally predict how many times the loop executes, or whether it will terminate. The key ingredient to prove the correctness of a loop is to provide the **loop invariant** (using the *invariant* clause) that is true immediately before and immediately after each iteration of the loop. The general loop invariant for the Boogie implementation is automatically generated. This is demonstrated on lines 21–22, i.e. we ensure that for each of the model elements that have been iterated upon, they have been resolved. Thus, by the end of the iteration, all the matched source elements are iterated and have been resolved, and therefore, the postcondition of the resolving algorithm can be established (lines 10–12).

Second, we use a **variant expression** to ensure that the loop terminates. A default variant expression (automatically generated) for the *ITERATE* EMFTVM instruction corresponds to the size of the iterated collection minus the corresponding loop counter (line 23 of Fig. 10). Since the counter increases on each iteration and the size of the processed collection remains unchanged, we can deduce that there are less elements in the collection to be iterated upon. In addition, since the loop counter has to be smaller than the length of the iterated collection (to keep the loop iterating), the variant expression is maintained above a bound (i.e. a lower bound of zero) so that it does not decrease forever. Note that the *decreases* clause on line 23 is not actually supported in Boogie. It is only used to demonstrate the concept of the variant expression. In practice, we record the old value of the *decreases* clause when entering the loop. Then, right after the corresponding counter of the loop increases, we check that: (a) It is greater than the current value of the *decreases* clause, and (b) It is greater or equal to the lower bound of zero.

Similarly, a variant expression is needed to ensure that the recursive call in the resolving algorithm terminates (line 2). Generally, variant expressions ensure that the call trace of the recursive call follows a predefined lexicographical metric [46]. We enforce the constraint that any call between mutually recursive calls leads to a strictly smaller metric value. In doing such a comparison, the input values of recursive calls are compared. For example, in our case, noticing the recursive call to the resolving algorithm on line 33, we need to ensure that the length of the input collection is decreasing

```
1  procedure resolve(this: ref, obj: ref) returns (r: ref);
2    decreases size(obj);
3    modifies tempHeap;
4    ...
5    ensures dtype(obj) = class.array ⟹
6      dtype(r) = class.array;
7    ensures dtype(obj) = class.array ⟹
8      size(Array2Seq(srcHeap, obj)) =
9        size(Array2Seq(tempHeap, r));
10   ensures dtype(obj) = class.array ⟹
11     (∀ j: int • 0≤j ∧ j<size(Array2Seq(srcHeap, obj)) ⟹
12       isResolved(Array2Seq(srcHeap, obj)[j]));
13   ensures dtype(obj) ≠ class.array ∧ ...
14       ⟹ (old(tempHeap) = tempHeap);
15
16 implementation resolve(this: ref, obj: ref) returns (r: ref)
17 {
18 ... /* the following code correspond to lines 21 - 27 shown in Fig.9 */
19 while(i<size(obj))
20   ...
21   invariant (∀ j: int • 0≤j ∧ j<i ⟹
22     isResolved(obj[j]));
23   decreases size(obj) - i;
24   {
25         stk := Seq#Build(stk, obj[i]);
26         /* 22: STORE lv2 */
27         call stk, e := OpCode#Store(stk);
28         /* 23: GETENV */
29         call stk := OpCode#GetENV(stk);
30         /* 24: LOAD lv2 */
31         call stk := OpCode#Load(stk, e);
32         /* 25: INVOKE resolve() */
33         call e_resolved := resolve(this, e);
34         stk := stk[..size(stk)-2] ++ e_resolved;
35         assume isResolved(e);
36         /  26: INVOKE including() */
37         call stk := Sequence#Including(stk);
38         /* 28: ENDITERATE */
39         i := i+1;
40   }
41 ...
42 }
```

Fig. 10 Example: reliable prototyping for the resolving algorithm of ATL (shown in Fig. 9) by encoding both its execution semantics and EMFTVM implementation in Boogie

with each call. Thus, we can deduce that there are less elements to be processed each time the resolving algorithm is recursively called. This property is obviously true for well-formed models stored in a tree-like persistent layer such as XMI (which requires input model to be strong containment [36]).

By encoding the execution semantics of the resolving algorithm and its corresponding EMFTVM bytecode implementation into Boogie, we can automatically verify that our prototype resolving algorithm is correctly implemented. Thus, we have confidence that the encoded execution semantics of the resolving algorithm is implementable and can be used to provide a sound verification of the MT. The complexity metrics for verifying the resolving algorithm are defined and evaluated in Sect. 6.

5 Translation validation for model transformation languages

When an MT language already has an EMFTVM implementation, we can use our formalisation for the EMFTVM bytecode language to represent the runtime behaviour of

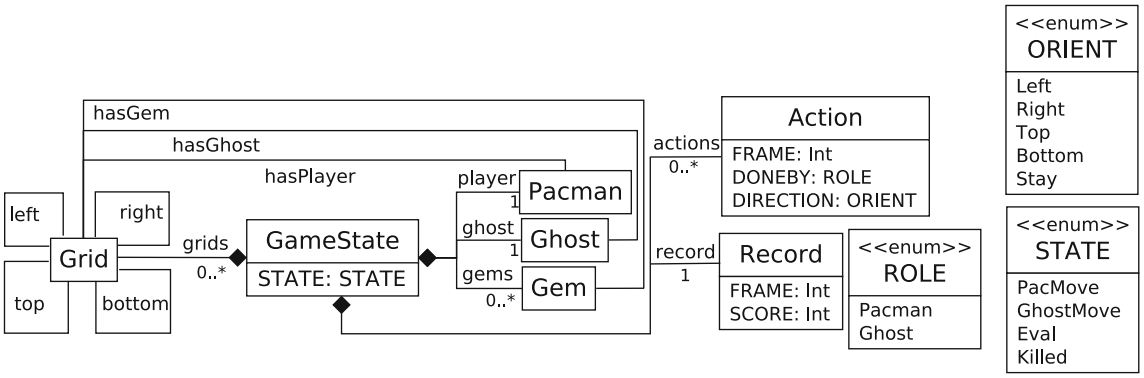

Fig. 11 Example: Pacman metamodel

its EMFTVM implementation. Thus, when we have a proposed execution semantics for the same MT language (e.g. by reverse engineering the MT language specification), we can verify that the proposed execution semantics for the MT language soundly represents the runtime behaviour of its underlying EMFTVM implementation. The benefit is that a MT verification that is based on such a translate–validate execution semantics is sound [19].

We identify the following steps to perform the translate validation approach for the target MT language:

- **(TV1)** Encoding the execution semantics of the MT language into Boogie.
- **(TV2)** Encoding the runtime behaviour of an existing EMFTVM implementation of the MT language into Boogie using our EMFTVM formalisation and then validating that the encoded execution semantics of the MT language is sound with respect to the runtime behaviour of its corresponding EMFTVM implementation in Boogie.

Currently, MT languages that target EMFTVM exist for ATL and SimpleGT [69]. In this section, to demonstrate the explicit model deallocation aspect of EMFTVM bytecode language, we focus on the SimpleGT graph transformation (GT) language. Before diving into the details of our translation validation approach, we first give a brief introduction to the SimpleGT language.

5.1 Overview of the SimpleGT Language

SimpleGT is an experimental GT language based on double push-out semantics developed by Wagelaar et al. [69]. A SimpleGT program is a declarative specification that documents what the SimpleGT transformation intends to do. It is expressed in terms of a list of rewrite rules, using OCL for both its data types and its declarative expressions. Then, the SimpleGT program is compiled into an EMFTVM implementation which can be executed.

Example 5.1 We use the Pacman game adapted from [60] to introduce the SimpleGT language. The game is based on the Pacman metamodel as shown in Fig. 11. The game consists of a single *Pacman*, a *ghost* and zero or more *gems* on a game board (consisting of more than zero *grids*). Each grid can hold Pacman, a ghost and a gem at the same time. The Pacman game is controlled by the *GameState*, which records important attributes such as *STATE*, *SCORE* and *FRAME*. It also contains a list of actions. Each action defines the moves permitted 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 (a snippet of this transformation is shown in Fig. 12). We provide 10 rules to move Pacman and the ghost in different directions (5 rules for each role), and we ensure that Pacman moves before the ghost. However, the evaluation (i.e. *Kill* or *Collect* rule) takes place after both of them have moved. Pacman collects a gem if both the gem and Pacman share the same grid. Pacman is killed by the ghost if both of them share the same grid. Finally, the GameState is updated by the *UpdateFrame* rule.

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

Take the *PlayerMoveLeft* rule of Fig. 12 for example; its input pattern specifies that:

```
1  module Pacman;
2
3  rule PlayerMoveLeft{
4    from
5      s : P!GameState(STATE=~PacMove, record=~rec),rec: P!Record,pac: P!Pacman,
6      grid2: P!Grid,grid1: P!Grid(hasPlayer=~pac, left=~grid2),
7      act : P!Action(DONEBY=~Pacman, FRAME=~rec.FRAME, DIRECTION=~Left)
8    not grid2: P!Grid(hasEnemy=~ghost), ghost: P!Ghost
9    to
10     s : P!GameState(STATE=~GhostMove, record=~rec),rec: P!Record,pac: P!Pacman,
11     grid2: P!Grid(hasPlayer=~pac),grid1: P!Grid(left=~grid2)
12 }
13 ...
```

Fig. 12 Example: excerpt of a SimpleGT transformation for *Pacman* model transformation

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

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

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

The implicit correspondence graph of the *PlayerMoveLeft* rule is calculated. Thus, what must be deleted (i.e. the difference between the input pattern and the correspondence pattern) is the value of *STATE* for game state *s*, the value of *hasPlayer* for *grid1*, the *act* and all the values of its structural features. What must be created (i.e. the difference between the output pattern and the correspondence pattern) is the value of *STATE* for the game state *s* that becomes *GhostMove*, and the value of *hasPlayer* for *grid2* that is set to *Pacman*.

In addition, unlike GT languages with explicit flow control (e.g. Henshin [3]), SimpleGT follows an automatic "fall-off" rule scheduling, i.e. if no match is found for a particular GT rule, it falls off to match the next rule.

5.2 TV1: execution semantics of SimpleGT

The execution semantics of rule scheduling in SimpleGT requires that it should be able to match rules with their own output, i.e. re-matching after each apply[4]:

– 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 initialising output elements as specified in the *to* section of the rule (*apply step*).
– After each application, the rule scheduling restarts immediately.
– When all rules have been processed (i.e. there are no more matches found for any rules), the rule scheduling stops.

The rule scheduling for SimpleGT is executed in *automatic recursive* mode by the EMFTVM virtual machine, which is different from the other modes in the EMFTVM virtual machine (e.g. the declarative aspect of the ATL language is executed in *automatic single* mode) [69]. This rule scheduling for SimpleGT implies that the source and target models are the same. Thus, there will be only one *heap* in our Boogie encoding.

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

– The first sub-step performs a structural pattern matching (by applying a search plan strategy [65]), where all the patterns that match the specified model elements and their structural relationship (i.e. an edge between model elements) are found. A subtlety here is that SimpleGT requires injective matching, i.e. all the model elements in each matched structural pattern are unique.
– The second sub-step is to iterate on the matched structural patterns for semantic pattern matching, where a pattern that satisfies specified semantic constraints is found (i.e. the constraints on the attributes of model elements given by the matching operator, and any NACs).

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

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

[4] For simplicity, we do not consider the inheritance of transformation rules [69].

```
1  procedure match_PlayerMoveLeft() returns (res: Seq ref)
2  ensures res=[] ⟹
3    (∀ i: int • 0≤i<size(findPatterns_PlayerMoveLeft(srcHeap))) ⟹ ¬(
4    // s : pacman_GameState(STATE=˜PacMove)
5    read(srcHeap,findPatterns_PlayerMoveLeft(srcHeap)[i][0],
6      pacman_GameState.STATE) = 3
7    ∧ ...);
8  ensures res≠[] ⟹
9    res ∈ findPatterns_PlayerMoveLeft(srcHeap)
10    ∧ read(srcHeap,res[0],pacman_GameState.STATE) = 3
11    ∧ ...);
12
13 function findPatterns_PlayerMoveLeft(): Seq< Seq<ref> >;
14 ... // Boogie axioms for the structural pattern matching.
```

Fig. 13 Example: encoding the execution semantics of the *match* step for the *PlayerMoveLeft* rule of *Pacman* transformation in Boogie

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

Example 5.2 A snippet of the Boogie encoding for the *match* step for the *PlayerMoveLeft* rule is shown in Fig. 13. The encoding conforms to the structure of the Boogie contract for the *match* step:

– If the result is an empty sequence, then in the source model, none of the pattern that passes the structural pattern matching (specified by the *findPatterns_PlayerMoveLeft* function) satisfies the constraints of semantics pattern matching (lines 2–7), e.g. the game is in a state *s* when *Pacman* should move (lines 4–6).
– If the result is not an empty sequence, then the result is a pattern (contains a sequence of source elements) in the source model that passes the structural and semantic pattern matching (lines 8–11).

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

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

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

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

The Boogie encoding for the *apply* step of the SimpleGT rules offers no surprise, as it strictly follows its execution semantics. Thus, we omit them in this article, and examples can be found in our online repository [20].

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

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

Fig. 14 Example: translation validating the soundness of our Boogie encoding for the execution semantics of the *match step* of the *PlayerMoveLeft* rule

validation approach to verify the soundness of our Boogie encodings for the execution semantics of SimpleGT.

5.3 TV2: translation validation of SimpleGT rules

Each SimpleGT rule is actually compiled into two EMFTVM code blocks by the EMFTVM compiler, i.e. a *match block* (for the *match step*) and an *apply block* (for the *apply step*). We have developed a code generator to automatically encode these EMFTVM code blocks into Boogie based on our EMFTVM formalisation.

In this section, we briefly describe our translation validation approach to verify the soundness of our Boogie encodings for the execution semantics of SimpleGT, i.e. that our Boogie encoding for the execution semantics of SimpleGT soundly represents the corresponding runtime behaviour given by the EMFTVM implementation.

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

(C1) the Boogie contract that represents the execution semantics of its *match step* is satisfied by the Boogie implementation that represents the runtime behaviour of its *match block*, and

(C2) the Boogie contract that represents the execution semantics of its *apply step* is satisfied by the Boogie implementation that represents the runtime behaviour of its *apply block*.

Each of these conditions forms a verification task that is handled by the Boogie verifier. If none of the verification tasks generate any errors (from the verifier), we conclude that our Boogie encoding for the execution semantics of the SimpleGT rules is sound. Otherwise, the trace information from the Boogie verifier, indicating where the encoding unsoundness was detected, will be output. Our evaluation in Sect. 6 confirms that we can automatically verify the soundness of each SimpleGT specification/EMFTVM implementation pair.

Example 5.3 We demonstrate our approach on the *match step* of the *PlayerMoveLeft* rule (Fig. 14). The runtime behaviour of its corresponding *match block* is implemented as a Boogie implementation (lines 5–12). The implementation contains two important sub-steps.

The first sub-step performs a structural pattern matching (line 8), where all the patterns that match the specified model elements and their structural relationships (i.e. edges between model elements) are found. Structural pattern matching is primarily implemented in Java instead of EMFTVM. Thus, they are only axiomatised using Boogie axioms (line 15) and are

not validated by the translation validation approach. This is a modular verification strategy, thus initiating the verification of the implementation for the SimpleGT transformations: we aim to verify each sub-component of MT language implementation individually. Consequently, we can clearly state which parts of the implementation are verified, which brings us a step closer to fully verify the MT language implementation.

The second sub-step is to iterate on the matched structural patterns (lines 9–12) for semantic pattern matching, where a pattern that satisfies the specified semantic constraint is found (i.e. the constraint on attributes of model elements given by the matching operator). The runtime behaviour of semantic pattern matching is given as a Boogie implementation (lines 22–29) written in terms of the translational semantics of EMFTVM. It is validated against the Boogie contracts for semantic pattern matching (lines 17–20) to ensure the soundness of its encoding.

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

Finally, we can conclude that the execution semantics of a SimpleGT specification encoded in Boogie is sound, when the execution semantics of both *match* and *apply* steps of all the relevant SimpleGT rules encoded in Boogie are sound (as defined by the soundness conditions *C1* and *C2*).

6 Evaluation of our reliable prototyping and translation validation approaches

In this section, we give implementation details to perform our approach (i.e. reliable prototyping and translation validation). We also evaluate the feasibility and performance of our approach on four case studies. The section concludes with a discussion of the obtained results and lessons learnt. We refer to our online repository for the generated Boogie programs for these case studies [20].

6.1 Implementation

To effectively evaluate our approach, we have implemented three kinds of code generators to automatically generate Boogie code:

- A code generator for EMF metamodels (*EMF2Boogie*), which generates the corresponding Boogie types and constants [21].
- Code generators for model transformation languages (e.g. *ATL2Boogie* and *SimpleGT2Boogie*), which generate Boogie procedures that represent the execution semantics of model transformations.
- A code generator for the EMFTVM bytecode language (*EMFTVM2Boogie*), which generates Boogie implementations that represent the runtime behaviour of transformation implementations.

Our translation validation approach interacts with all three kinds of code generators. Our reliable prototyping approach interacts with the third code generator only. Notice that our case studies (Sect. 6.2) focus on the ATL and SimpleGT languages. Thus, when applying our translation validation to other MT languages, a code generator of the second kind needs to be implemented. We envision that our code generators for the ATL and SimpleGT languages could provide an example for implementing such code generators.

Template-based model-to-text tools are used to implement the first two kinds of code generators (e.g. XPand [40], XTend [11]). Generally, these code generations start by serialising the input into models (using tools provided by the input language, e.g. the ATL extractor API provided by the ATL compiler). Then, the models generate the corresponding Boogie code according to the templates we defined.

The EMFTVM2Boogie code generator is implemented in Java. This is because in our experience the logic and computations involved in the EMFTVM2Boogie code generator are more intuitive to express in Java than with the template-based model-to-text tools. The general idea of implementing the EMFTVM2Boogie code generator is to read in each code block of the input EMFTVM implementation. Then, according to our Boogie library for the formalisation of the EMFTVM bytecode language (Sect. 3), we generate the corresponding Boogie code for each EMFTVM instruction in each code block. Specifically, our Boogie library for the formalisation of the EMFTVM bytecode language consists of:

- A list of Boogie procedures to encapsulate the operational semantics of EMFTVM instructions, e.g. the *PUSH* instruction.
- Code generation knowledge, written in terms of a comment (for documentation purposes only), for the EMFTVM instructions whose operational semantics is not suitable for encapsulation by the Boogie procedure. In our opinion, those instructions are not suitable to be encapsulated because their encapsulation will either be *high order* (e.g. *IF* instruction) or will make the code generation unnecessarily complex (e.g. the *ADD* instruction has to manage its frame condition when it is encapsulated as a Boogie procedure).

In addition to this, the main challenge in developing EMFTVM2Boogie stems from the fact that each EMFTVM block is based on a generic operand stack. This would compromise the precision of our generated Boogie implementations. For example, a *GET name* instruction simply

retrieves the *name* attribute for the top model element on the operand stack. However, because we use separate heaps to represent the input and output models, the type of the top model element on the operand stack is important for the EMFTVM2Boogie in generating corresponding Boogie code. Thus, we also maintain a type stack during the code generation for each EMFTVM instruction. For example, the *PUSHI* instruction has the effect of pushing an integer onto the type stack, so that its next instruction can look up the type stack and query its state.

6.2 Evaluation setup

The goal of our evaluation is to quantitatively assess the performance and feasibility of our approach. More specifically, we aim to answer the following questions:

Q1: *Feasibility* Are our reliable prototyping and translation validation approaches feasible to apply to model transformation languages? If yes, is there any evidence shown as the outcome of this feasibility?
Q2: *Performance* Can our reliable prototyping and translation validation approach be performed in a time-efficient and automatic manner? If not, what is the reason for this?

Therefore, we set up 4 case studies to answer these questions:

- (Resolving) The resolving algorithm that we reliably prototyped, which is crucial to the performance and viability of common trace-based MT languages.
- Two case studies which demonstrate our translation validation approach for the declarative aspect of the ATL language (i.e. ATL matched rules with default scheduling against normal/abstract classifiers):
  - (ER2REL) The *ER2REL* transformation that translates an ER diagram to a relational schema. It is a modified version of the one originally developed by Büttner et al. [15]. The modification does not cause the ATL transformation to behave differently. However, it contains a feature (i.e. consecutive bindings in an ATL matched rule) that is not considered in the previous work.
  - (HSM2FSM) The *HSM2FSM* transformation that translates a hierarchical state machine to a flattened state machine. This was originally presented by Baudry et al. [8] to demonstrate the challenges in MT testing.
- (Pacman) The Pacman transformation that gives the operational semantics of the Pacman game. This case study demonstrates our translation validation approach between the SimpleGT language (double push-out semantics with NAC, injective matching, and automatic "fall-off" rule scheduling).

Table 1 The transformation complexity metrics of our case studies

	ER2REL	HSM2FSM	Pacman
Metamodel metrics (source → target)			
No. of classifiers	5 → 3	6 → 6	7 → 7
No. of attributes	6 → 4	2 → 2	6 → 6
No. of associations	6 → 2	5 → 5	13 → 13
Transformation metrics			
No. of rules	6	7	13
No. of rule filters	3	5	40

The evaluation of first and fourth case studies is performed according to the description given in Sects. 4 and 5, respectively. The evaluation of second and third case studies is performed as follows:

(TV1) Encoding the execution semantics of the ATL transformation under study into Boogie.
(TV2) Encoding the runtime behaviour of the EMFTVM implementation of the corresponding case study into Boogie using our EMFTVM formalisation and then validating that the encoded execution semantics is sound with respect to the runtime behaviour of its corresponding EMFTVM implementation in Boogie.

We reuse the execution semantics of the ATL transformations from our previous work, which is specific to ATL transformations that are compiled to the ASM implementations [19]. Thus, the design of second and third case studies intends to show that different transformation implementations for the ATL language agree on the same execution semantics.

As shown in Table 1, we choose 5 metrics, from the metrics developed by Vepa et al. and Vignaga, to measure the transformation complexity of our case studies [66,67]. These 5 metrics do not apply to the first case study (since it is a transformation algorithm rather than a MT). The first three metrics in Table 1 measure the quantity of metamodel constructs involved in the MTs. The 4th and 5th rows quantify the complexity of the MT in terms of the number of transformation rules involved and rule filters specified.

Finally, our evaluation uses the Boogie verifier (version 2.2) and Z3 (version 4.3). It is performed on an Intel 3.7 GHz machine with 8 GB of memory running the Windows operating system. Our evaluation results are represented in the next Section.

Table 2 The verification complexity metrics of our case studies

	Resolving	ER2REL	HSM2FSM	Pacman
Results (verified/total)	1 / 1	11 / 12	14 / 14	18 / 26
Veri. time (seconds)	0.004	0.030	0.074	0.159
Boogie (lines of code)	250	1033	1402	5018
Automation?	No	Yes	Yes	Yes
Termination?	Yes	Yes	Yes	N/A

6.3 Evaluation results

Table 2 measures the verification complexity of the four case studies. The verification results are recorded in terms of verified and total generated verification tasks for each case study.[5] Verification times are recorded in seconds. The lines of Boogie code generated for each case study include Boogie encodings for the metamodels, the execution semantics of the MT and the corresponding runtime behaviour of the EMFTVM implementation. "Automation" is measured by whether human interaction is involved during the verification. The "Termination" column shows whether the case study has terminated.

Q1: Feasibility As shown in Table 2, it is feasible to apply our reliable prototyping and translation validation approach to model transformation languages. However, our case study for reliable prototyping covers just one single feature for model transformation languages, which is remote from prototyping an entire model transformation language reliably. Thus, we cannot claim all model transformation features can be designed using our reliable prototyping approach. This is related to the expressiveness of our approach, which we discuss further in Sect. 6.4.

Moreover, as shown in Table 2, not all verification tasks of the ER2REL transformation have been verified. For one of the rules in the ER2REL transformation, our translation validation approach reports that its execution semantics (applying step) is not sound with respect to the runtime behaviour of its corresponding EMFTVM implementation. This is related to the consecutive bindings used in this rule: our execution semantics interprets them as a *composition* of the two bindings (which is verified to be sound with respect to the runtime behaviour of the ASM implementation [19]). However, the EMFTVM implementation interprets them as the second binding *overwrites* the first binding.

Thus, our evaluation concludes that the ASM and EMFTVM transformation implementations for the ATL language do not agree on the same execution semantics. The fundamental reason for this runtime behaviour mismatch is that the semantics of ASM bytecode language is not aligned with the EMFTVM bytecode language. For example, the *SET* instruction in the ASM bytecode language behaves polymorphically according to its argument [19]. However, it behaves monomorphically to its argument in the EMFTVM bytecode language (Appendix).

In addition, as shown in Table 2, eight verification tasks of the Pacman transformation have not been verified. On investigation, we found that these verification tasks have not failed because the execution semantics of Pacman transformation is not sound with respect to the runtime behaviour of its corresponding EMFTVM implementation. Rather, they fail because the Pacman transformation contains eight bindings that potentially could cause runtime exceptions, caught by the formalisation of *ADD* instruction in Boogie (Sect. 3.4).

For example, in the output pattern of the *PlayerMoveLeft* rule of Fig. 12, the *hasPlayer* could potentially be added twice to the *grid2* (since we have not checked in the NACs about whether the *hasPlayer* of *grid2* is set or not) and cause a runtime exception. However, recall that a well-formed input model should contain a single *Pacman* and a *ghost* (Sect. 5.1). Therefore, any well- formed input model would not cause such runtime exception. After we add a precondition to establish that the inputs of Pacman transformation are well formed, all the verification tasks are verified.

Thus, our evaluation confirms the feasibility of using our formalisation of EMFTVM bytecode to check the absence of runtime exceptions and improve the reliability of developed model transformations.

Furthermore, we identify two reasons for the compatibility of the translation validation approach with MT verification. First, the translation validation approach is inherently efficient. That is, the soundness encoding only needs to be verified once for each compilation to ensure the encoded execution semantics of each MT soundly represents the runtime behaviour of its corresponding implementation. Such soundly encoded execution semantics of MTs can be reused in order to verify the correctness of MTs against their specified contracts, as long as the source MT does not change.

The second reason is due to the features of the target MT language. Take the ATL language for example:

1. Each ATL rule is written in a declarative style and has a unified deterministic goal to achieve (i.e. mapping). Thus, it is easier to abstract the semantics of ATL rules into FOL expressions than to abstract the semantics of an imperative program that achieves an arbitrary goal. This feature greatly reduces the complexity of adapting the translation validation approach and enables its automation.
2. We consider ATL matched rules in non-refinement mode, which are always propagated on an initially empty target model that is disjoint from the source model. Thus, we are able to use two separate *heaps* to organise the source and

[5] We count each generated Boogie procedure/implementation pair as a verification task.

target elements. This ensures, for example, a modification made on the target *heap* will not affect the state of the source *heap*. Therefore, it yields a simple encoding that contributes to the automation of the translation validation approach.

3. For ATL matched rules, any iteration in the bytecode implementation always interacts with collections. Thus, we are able to automatically infer suitable *invariant* and *variant* expressions for loops. This feature is generally not obtainable for general programming languages, where iteration can loop over a user-defined data structure, e.g. a linked list.

Q2: Performance As shown in Table 2, we conclude that both our reliable prototyping and translation validation approach can be performed in a time-efficient manner. However, the verification process of the prototyped resolving algorithm requires guidance from the user. The difficulty stems from the need to manually construct a proper variant function (for recursive calls) to prove its termination (Sect. 4.3). How to automate such tasks is beyond the scope of this paper, as it would require advanced verification techniques such as abstract interpretation [23], to be adapted in a MT verification context.

Termination While we can successfully verify the termination of *ER2REL* and *HSM2FSM* transformations, we cannot verify the termination of *Pacman* transformations. The main challenge stems from finding an appropriate variant function for its rule scheduling. Thus, we plan to investigate other approaches for verifying termination of the GT language in the future (e.g. coloured Petri nets [33]).

6.4 Limitations of our approach

The evaluation results strongly demonstrate the feasibility and performance of our approach. However, our current approach has some limitations.

6.4.1 Soundness

Soundness prevents false negatives. The soundness of our approach depends on the consistency of our Boogie library for the formalisation of the EMFTVM bytecode language. At the moment, our Boogie library is structural, intuitive and available for inspection. In addition, we have designed a regression test suite with test oracles that specifies verification scenarios and their expected outcome. The regression test suite is executed on every modification to our Boogie library, to ensure the soundness of our approach on each rebuild.

6.4.2 Completeness

Completeness prevents false positives. The incompleteness of our approach could be due to known limitations of underlying SMT solvers when working with quantifiers [29,32,47]. Boogie allows triggers (a.k.a matching patterns) to declare how to instantiate quantifiers, which is crucial to the completeness and performance of the SMT solver. Leino and Monahan describe how to use triggers effectively in the axiomatisation of summation-like comprehensions in Boogie [45], which is the guidance we followed in our approach. Potential incompleteness could also be due to missing axioms in our Boogie libraries. For example, our Boogie library for EMFTVM formalisation encodes only the essential axioms required to define its meaning. The auxiliary axioms such as "*POP* immediately after *PUSH* instruction would not effect the operand stack" are not in our encoding. We think it is better to let the users of our Boogie libraries decide when to introduce the auxiliary axioms. Consequently, we can maintain a set of essential axioms in our Boogie libraries, which facilitates manual inspection and reduces the possibility of inconsistent axioms.

6.4.3 Expressiveness

Because of the underlying SMT solver, the expressiveness of our approach is based on FOL with equality. To ensure this expressiveness power is useful for our approach in practice, we need to experiment with more MT languages.

6.4.4 User experience

On verification failure, the trace information from the Boogie verifier, indicating where the contract violation (within the input Boogie program) was detected, will be output. In our experience, such information is very useful when debugging the Boogie program by inserting additional assertions to understand it better, getting closer to the exact issue that causes the verification failure. However, we admit this is currently a limitation of our approach, since we are not capable of reporting useful feedback to a non-Boogie expert on verification failure. A potential solution is to find a concrete counter-example in terms of a transformation scenario to reproduce the verification failure (e.g. by model finding [15,73]).

7 Related work

There is a large body of work on the topic of ensuring the correctness of MTs. These include survey articles [1,17,55]. In this section, we primarily focus on the literature that falls

within the scope of the formal verification of MTs. We categorise these articles by the formal method applied.

7.1 Simulation, model checking and model finding

Simulation approaches require that a mathematical model be developed. This mathematical model represents the key characteristics of the MT (e.g. source and target metamodels, the behaviour of the MT 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 correctness properties can be expressed as contracts and can then be verified for the input.

For example, coloured Petri nets have been used to simulate Query/View/Transform (QVT) MTs. Thus, a coloured Petri net engine can be used to check contracts such as termination [33,72]. Similarly, Troya and Vallecillo use rewriting logic to simulate ATL MT in the Maude system [63]. Their system allows a reachability analysis of the ATL. Syriani and Vangheluwe propose an input-driven simulation approach using the Discrete EVent system Specification (DEVS) formalism [60].

Similarly to simulation approaches, model checking and model finding approaches require that a mathematical model be developed (from the metamodels and the MT specification). However, no particular input is needed when the model checking/finding is executing.

A subtle difference between the model checking and model finding approaches is in the way that they use the developed mathematical model [35]. The former starts with a mathematical model described by the user and discovers whether the contracts asserted by the user are valid on the mathematical model. The later 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 [48,49]. 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 MTs is always finite.

Anastasakis et al. [2] have designed the UML2Alloy tool to perform model finding. The novelty of their work is the use of Alloy, which is a verification language for SAT-based model finding [37]. Anastasakis et al. use Alloy as an IVL to ease (i) the encoding of MOF metamodels (enriched with syntactic correctness contracts expressed in OCL) and MTs to Alloy; (ii) the generation of SAT formulas from Alloy. Jackson et al. [36] have designed the Formula framework, which is based on the Z3 SMT solver [28]. Their main contribution is the systematic encoding of MOF metamodels and MT specifications using algebraic data types. The contracts are written in FOL. Consequently, they can use their framework to find models that witness violations of syntactic correctness in the given MT specification.

7.2 Theorem proving

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

Combemale et al.[22] present a pen-and-paper bisimulation proof to show that the ATL MT generates a Petri net model that preserves the observational operational semantics of an xSPEM model. Calegari et al. [16] encode the ATL MT and its metamodels into inductive types. 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 MT 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 [18,42,53,54]. This is the opposite of traditional theorem-proving approaches. At its simplest, the idea is to represent the metamodels as terms. Then, each MT 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 MTs by construction [42]. It uses a combination of UML and OCL to create a MT design, instead of using types. UML use case diagrams and activity diagrams are used to graphically create a MT specification. The specification is optionally constrained by OCL contracts on source and target metamodels. Then, the MT specification is verified against its contracts by translating both into abstract machine notations in B or as input for the Z3 SMT solver for theorem proving. Finally, the verified MT design can be synthesised to an executable transformation implementation (such as a Java program or an ATL transformation).

Asztalos et al. [4] use category theory to describe graph rewriting systems. This approach is implemented in the VMTS verification framework, but it is not targeted to a specific graph rewriting-based MT language. Schätz [57] presents an approach to verify the structural contracts of GT. The transformation rules are given as a textual description based on a relational calculus. The formalisations of model, metamodel and transformation rules are based on declara-

tive relations in a Prolog style and target the Isabelle/HOL theorem prover.

Büttner et al. [15] automate the process of theorem proving by a novel use of SMT solvers. 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.

7.3 Discussion

All of the approaches that we have just described rely on encoding the execution semantics of the MT language. However, existing approaches do not verify that the encoded execution semantics soundly represents the runtime behaviour provided by the implementation. Therefore, an unsound encoding will yield unsound results after verification, i.e. it will lead to erroneous conclusions about the correctness of the MT. In a MT 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 [55].

Therefore, we address this challenge by a novel usage of the translation validation approach [52], to verify that this execution semantics soundly represents the runtime behaviour of its corresponding EMFTVM implementation. It thus makes our approach complementary to the existing approaches.

Translation validation is one of the techniques used in compiler verification. Instead of formally proving the compilation is correct over all legal input programs, the primary assumption of translation validation is that the validator (i.e. the tool to perform translation validation) has limited knowledge of the compilation implementation. Hence, a variety of methods for translation validation arise [52,61,62]. Compared to these methods, our translation validation approach mainly differs in the representation of source and target code, and the definition of a correct compilation. That is, we specifically interest in verifying that the execution semantics of the source MT produced on this run is sound with respect to the runtime behaviour of its corresponding transformation implementation.

We developed our approach in Boogie IVL. The two most widely used IVLs are Boogie [6] and Why3 [30]. Both of them are based on FOL with polymorphic types and have mature implementations to parse, type-check and analyse programs. We concentrate on Boogie in this research, but we believe all results can be reproduced in Why3.

8 Conclusions and future work

In this paper, we present a formalisation for the EMFTVM bytecode language. Our formalisation is modularised in the Boogie IVL as a library for reuse. It assists the modeller in developing the MT implementation correctly (reliable prototyping) and prohibits unimplementable execution semantics for the MT language from being introduced into a sound verification pipeline. It also enables a translation validation approach to ensure the encoded execution semantics correctly represents the runtime behaviour of the underlying MT language implementation. Our evaluation is performed on the resolving algorithm (crucial to the performance and viability of trace-based MT languages), ATL (one of the most widely used MT languages in industry and academia) and SimpleGT (an experimental GT language). The evaluation strongly demonstrates the feasibility of our approach.

In the future, we plan to prototype the EMFTVM implementation for a new scalable MT language that is dedicated to very large models. We believe an existing approach that is based on map-reduce could provide guidance for this task [9]. One of our foci is then to use our approach to reliably prototype our EMFTVM implementation and ensure it is verified against the proposed execution semantics for such a scalable MT language. In addition, we also anticipate the need to enrich the EMFTVM bytecode language for efficient distributed model manipulation.

Our current evaluation is designed to show the feasibility and performance of our approach. We plan to learn from the state of the art that applies mutation testing techniques to model transformations (e.g. [14,51,56]) to further evaluate the completeness of our approach.

Finally, we believe that the idea of automating the translation validation approach by language reduction could be adapted to other programming languages. The challenge is to find a balance between the level of automation needed and the degree of language reduction. This is a challenge we would like to investigate in the near future.

Acknowledgements Zheng Cheng is funded by the Doctoral Teaching scholarship and John & Pat Hume scholarship from Maynooth University and the MONDO (EU ICT-611125) Project. We thank the reviewers for their feedback to improve the representation of this submission.

Appendix: The translational semantics of EMFTVM bytecode language

In this section, we illustrate the translational semantics of the EMFTVM bytecode language in more detail. Overall, 42 out of 48 instructions are encoded by our formalisation for the EMFTVM bytecode language. The 6 EMFTVM instructions that are not encoded are omitted because of some technical

limitations of our approach (e.g. transitive closure cannot be described in FOL [36]).

Some of the conventions we use are:

- In general, we draw on the *let..in* expression to improve the readability of our formalisation. However, this is not a standard syntax in Boogie.
- We use the superscript # to denote the line number of an EMFTVM instruction. This superscript is attached to the declared Boogie variables to avoid name collision.
- We also use the notation of $[\![S]\!]$ to represent the transformation from the EMFTVM construct S to its corresponding Boogie code.

In addition to our general encoding convention, the auxiliary notations used by our formalisation for stack, collection, heap, method invocation and metamodel are explained in Table 3.

Table 3 Auxiliary notations used by the translational semantics of EMFTVM bytecode language

Auxiliary notation	Comment
Stack	
e ::l	Return the concatenation of the element *e* to the sequence *l*
l[low..upper]	Return the subsequence of *l* from index *lower* to *upper*
hd(l: Sequence)	Return the first element of the given sequence, which requires the input sequence is not an empty sequence
tl(l: Sequence)	Return the rest of the input sequence that excluding the first element, which requires the input sequence is not an empty sequence
tk(l: Sequence, n: int)	Return a new sequence which takes the first *n* elements from the input sequence
dp(l: Sequence, n: int)	Return a new sequence which drops the first *n* elements from the input sequence
size(l: Sequence)	Return the size of the given sequence
Collection	
hasNext(c: Collection)	Return *true* if the passed-in collection has more elements to iterate
next(c: Collection)	Return the next element of the collection in the iteration
isCollection(f: Field α)	Return *true* when the given structure feature is an association and its multiplicity has an upper bound that is greater than one, and return *false* otherwise
c1 $\cup$ c2	Return a new collection that appends the collection *c2* to the collection *c1*
c1 - c2	Return a new collection such that every element of *c2* are removed from the collection *c1*
Heap	
read(h: heap, o: ref, f: Field α)	Return the value of the heap *h* at the position given by the element *o* and the structural feature *f*
update(h: heap, o: ref, f: Field α, v: α)	Change the value of the heap *h* at the position given by the element *o* and the structural feature *f*, to the value of *v*
dtype(o: ref)	Return the classifier of the input element

Table 3 continued

Auxiliary notation	Comment
Method Invocation	
reflect(sig:String)	Return the method with the given signature name
invoke(mtd: Method, args: Sequence)	Return the invocation result of the input method on the given arguments
Metamodel	
resolve(mm: String, cl: String)	Return the corresponding classifier resolved from the input metamodel and classifier name
toRef(cl: String)	Return an unique Boogie constant of type *ref* for the input classifier name

Table 4 Translational semantics for EMFTVM stack handling instructions

EMFTVM instruction (S)	Corresponding Boogie statements (⟦S⟧)
PUSH c	stk := ⟦c⟧ :: stk;
PUSHT	stk := `true` :: stk;
PUSHF	stk := `false` :: stk;
POP	`assert` size(stk) > 0; stk := tl(stk);
STORE x	`assert` size(stk) > 0; ⟦x⟧ := hd(stk); stk := tl(stk);
LOAD x	stk := ⟦x⟧ :: stk;
SWAP	`assert` size(stk) > 1; stk := hd(tl(stk)) :: hd(stk) :: tl(tl(stk));
SWAP_X1	`assert` size(stk) > 2; stk := hd(tl(tl(stk))) :: hd(stk) :: hd(tl(stk)) :: tl(tl(tl(stk)));
DUP	`assert` size(stk) > 0; stk := hd(stk) :: stk;
DUP_x1	`assert` size(stk) > 1; stk := hd(stk) :: hd(tl(stk)) :: hd(stk) :: tl(tl(stk));
NOT	`assert` size(stk) > 0; stk := (¬(hd(stk))) :: tl(stk);
AND Stmt	`assert` size(stk) > 0; ⟦Stmt⟧ stk := (hd(tl(stk)) ∧ hd(stk)) :: tl(tl(stk));
OR Stmt	`assert` size(stk) > 0; ⟦Stmt⟧ stk := (hd(tl(stk)) ∨ hd(stk)) :: tl(tl(stk));
XOR	`assert` size(stk) > 1; stk := ((hd(stk) ∨ hd(tl(stk))) ∧ ¬(hd(stk) ∧ hd(tl(stk)))) :: tl(tl(stk));
IMPLIES Stmt	`assert` size(stk) > 0; ⟦Stmt⟧ stk := (¬(hd(tl(stk))) ∨ hd(stk)) :: tl(tl(stk));
ISNULL	`assert` size(stk) > 0; stk := (hd(stk)=null) :: tl(stk);
GET_CB Stmt	LABEL: ⟦Stmt⟧
GET_TRANS	currently not supported

The full translational semantics of the EMFTVM language is given in Tables 4, 5 and 6, classified by the category that each EMFTVM instruction resides in.

In general, our formalisation for the EMFTVM bytecode language is based on two main data structures (Sect. 3.1): the operand stack *stk* and the global memory model *heap*. We further introduce an access table *acc* to prohibit runtime exceptions caused by certain operations on the structural features (Sect. 3.4). Thus, checking whether *o.f* is set or not becomes an expression *isset*(*acc*, *o*, *f*). Marking *o.f* as set or not uses the expression *set(acc, o, f, v)*.

In addition to the discussion of our formalisation for the EMFTVM bytecode language in Sect. 3, we explain six additional points:

1. In Table 4, the operational semantics of the *GET_CB* instruction is to push the code block *Stmt* onto the operand stack. It usually works with the *INVOKE_CB_S* instruction (Table 5), whose operational semantics is to execute the code block that is on top of the operand stack. Thus, in the translation rule for the *GET_CB* instruction, we introduce a fresh label (denoted by *LABEL*) to identify the translated EMFTVM code block, which is referred by the translation rule for the *INVOKE_CB_S* instruction.
2. In Table 5, to make sure the offset of the *GOTO*, *IF* and *IFN* instructions is valid in their corresponding translation rules, we insert a fresh Boogie label, denoted by ⟦n⟧, at the program point which corresponds to the offset *n* of these instructions (Sect. 3.3).
3. In Table 5, the operational semantics of the *RETURN* instruction is to return to the *caller* code block that

Table 5 Translational semantics for EMFTVM control flow instructions

EMFTVM Instruction (S)	Corresponding Boogie Statements (⟦S⟧)
GOTO n	`goto` ⟦n⟧;
RETURN	`goto` END;

IF n	`var` cond# : bool;
	`assert` size(stk) > 0;
	cond# := hd(stk);
	stk := tl(stk);
	`if` (cond#) `goto` ⟦n⟧;

IFN n	`var` cond# : bool;
	`assert` size(stk) > 0;
	cond# := hd(stk);
	stk := tl(stk);
	`if` (¬cond#) `goto` ⟦n⟧;

IFTE Stmt1 Stmt2	`var` cond# : bool;
	`assert` size(stk) > 0;
	cond# := hd(stk);
	stk := tl(stk);
	`if` (cond#) ⟦Stmt1⟧ `else` ⟦Stmt2⟧

ITER Stmt **ENDITER**	`var` col# : Seq ref;
	`assert` size(stk) > 0;
	col# := hd(stk);
	stk := tl(stk);
	`while` (hasNext(col#))
	{stk := next(col#) :: stk;
	⟦Stmt⟧}

INVOKE sig n	let args = tk(stk, n) in `var` result# : T; `assert` size(stk) ≥ n; `call` result# := invoke(reflect(⟦sig⟧), args); stk := result# :: dp(stk, n);
INVOKE_STATIC sig n	let args = tk(stk, n) in `var` result# : T; `assert` size(stk) ≥ n; `call` result# := invoke(reflect([sig]), toRef(hd(args)) :: tl(args)); stk := result# :: dp(stk, n);

INVOKE_CB Stmt n	let args = tk(stk, n) in
	`var`stk# : Seqα;
	`assert` size(stk) ≥ n;
	stk# := stk;
	stk := args;
	⟦Stmt⟧
	END:
	stk := hd(stk) :: dp(stk#, n);

INVOKE_CB_S n	let args = tk(stk, n) in
	`var` stk# : Seq α;
	`assert` size(stk) ≥ n;
	stk# := stk;
	stk := args;
	goto LABEL;
	END :
	stk := hd(stk) :: dp(stk#, n);

INVOKE_SUPER	currently not supported
INVOKE_ALL_CBS	currently not supported

invokes the current code block. Thus, during the Boogie code generation, we insert a fresh Boogie label after each code block invocation, denoted by *END* as shown in the translation rules of the *INVOKE_CB* and *INVOKE_CB_S* instructions (Table 5). Thus, our translation rule for the *RETURN* instruction simply goes to the *END* label and thus has the effect of returning the execution to the caller code block.

4. In Table 6, the *OCLType#allInstance* function used by the translation rule for the *ALLINST* instruction comes from our Boogie library for OCL [19]. It returns a sequence of the currently allocated elements on the given *heap* whose classifier is as specified by the input classifier name.
5. In Table 6, the *default*, referred by the translation rule for the *REMOVE* instruction, is a shorthand to represent the default value of EMF types in Boogie, e.g. the default value for *boolean* type is *false*, for *integer* type is *0*, for

Table 6 Translational semantics for EMFTVM model handling instructions

EMFTVM Instruction (S)	Corresponding Boogie Statements (⟦S⟧)

NEW mm	let cl = hd(stk) in
	let clazz = resolve(⟦mm⟧, cl) in
	var $r^{\#}$: ref;
	assert size(stk) > 0;
	havoc $r^{\#}$;
	assume $r^{\#} \neq$ null $\wedge \neg$read(heap, $r^{\#}$, alloc) $\wedge$ dtype($r^{\#}$) = clazz;
	heap := update(heap, $r^{\#}$, alloc, true);
	stk := $r^{\#}$ :: tl(stk);

NEW_S	let mm = hd(stk), cl = hd(tl(stk)) in
	let clazz = resolve(mm, cl) in
	var $r^{\#}$: ref;
	assert size(stk) > 1;
	havoc $r^{\#}$;
	assume $r^{\#} \neq$ null $\wedge \neg$ read(heap, $r^{\#}$, alloc) $\wedge$ dtype($r^{\#}$) = clazz;
	heap := update(heap, $r^{\#}$, alloc, true);
	stk := $r^{\#}$ :: tl(tl(stk));

GET f	let o = hd(stk) in
	assert size(stk) > 0 $\wedge$ o $\neq$ null $\wedge$ read(heap, o, alloc);
	stk := read(heap, o, [f]) :: tl(stk);

SET f	let o = hd(tl(stk)), v = hd(stk) in
	assert size(stk) > 1 $\wedge$ o $\neq$ null $\wedge$ read(heap, o, alloc);
	heap := update(heap, o, ⟦f⟧, v);
	stk := tl(tl(stk));

GET_STATIC f	let cl = hd(stk) in assert size(stk) > 0; stk := read(heap, toRef(cl), ⟦f⟧) :: tl(stk);

SET_STATIC f	let cl = hd(tl(stk)), v = hd(stk) in
	assert size(stk) > 1;
	heap := update(heap, toRef(cl), ⟦f⟧, v);
	stk := tl(tl(stk));

FINDTYPE mm cl	stk := resolve(⟦mm⟧, ⟦cl⟧) :: stk;

FINDTYPE_S	let mm = hd(stk), cl = hd(tl(stk)) in
	assert size(stk) > 1;
	stk := resolve(mm, cl) :: tl(tl(stk));

ALLINST	let cl = hd(stk) in
	var $col^{\#}$: Seq ref;
	assert size(stk) > 0;
	$col^{\#}$:= OCLType#allInstance(heap, cl);
	stk := $col^{\#}$ :: tl(stk);

ALLINST_IN	let cl = hd(tl(stk)), hp=hd(stk) in
	var $col^{\#}$: Seq ref;
	assert size(stk) > 1;
	$col^{\#}$:= OCLType#allInstance(hp, cl);
	stk := $col^{\#}$ :: tl(tl(stk));

Table 6 continued

EMFTVM instruction (S)	Corresponding Boogie statements (⟦S⟧)

DELETE	let o = hd(stk) in
	var heap# : HeapType;
	heap# := heap;
	assert size(stk) > 0;
	assert o ≠ null ∧ read(heap, o, alloc);
	havoc heap;
	assume (∀r: ref, f: Field α •
	r ≠ null ∧ read(heap#, r, alloc) ∧ r ≠ o ⟹
	read(heap, r, f) = read(heap#, r, f));
	assume (∀r: ref, f: Field α •
	r ≠ null ∧ ¬read(heap#, r, alloc) ⟹
	read(heap, r, f) = read(heap#, r, f));
	assume (∀f: Field α •
	f ≠ alloc ⟹
	read(heap, o, f) = read(heap#, o, f));
	assume ¬read(heap, o, alloc);
	stk := tl(stk);

ADD 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
	{assert ¬isset(acc, o, [f]);
	heap := update(heap, o, [f], v);
	acc := set(acc, o, [f], true); }
	stk := tl(tl(stk));

REMOVE f	let o = hd(tl(stk)), v = hd(stk) in
	assert size(stk) > 1 ∧ o ≠ null ∧ read(heap, o, alloc);
	if (isCollection(⟦f⟧))
	{ heap := update(heap, read(heap, o, ⟦f⟧), read(heap, o, ⟦f⟧)- v); }
	else
	{ if(read(heap, o, ⟦f⟧) = v)
	{ assert isset(acc, o, ⟦f⟧);
	heap := update(heap, o, ⟦f⟧, default);
	acc := set(acc, o, ⟦f⟧, false); } }
	stk := tl(tl(stk));

INSERT f	let o = hd(tl(tl(stk))), v = hd(tl(stk)), i = hd(stk) in
	assert size(stk) > 2 ∧ o ≠ null ∧ read(heap, o, alloc);
	if (isCollection(⟦f⟧))
	{ assert − 1 ≤ i ∧ i < size(read(heap, o, ⟦f⟧));
	heap := update(heap, o, ⟦f⟧, read(heap, o, ⟦f⟧)[0..i-1] :: v ::
	read(heap, o, ⟦f⟧)[i + 1..size(read(heap, o, ⟦f⟧))-1]); }
	else
	{assert ¬ isset(acc, o, ⟦f⟧);
	heap := update(heap, o, ⟦f⟧, v);
	acc := set(acc, o, ⟦f⟧, true); }
	stk := tl(tl(tl(stk)));

GETENV	stk := ExecEnv :: stk;
GETENVTYPE	stk := dtype(ExecEnv) :: stk;
MATCH	currently not supported
MATCH_S	currently not supported
GET_SUPER	currently not supported

string type is an empty sequence and for any other types is *null*.

6. In Table 6, the *ExecEnv*, referred by the translation rule for the *GETENV* and *GETENVTYPE* instructions, is a Boogie constant of type *ref* that represents the execution environment of EMFTVM. Currently, our formalisation for the EMFTVM bytecode language does not provide axioms to encode its semantics. This technical limitation requires more thorough examination for the source code of EMFTVM, which we are currently working on. This leads to the absence of the translation rule for EMFTVM instructions such as *MATCH* and *MATCH_S*, since these instructions require static information from the execution environment of EMFTVM.

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: 5th International Conference on Software Testing, Verification and Validation. pp. 921–928. IEEE, Washington, DC, USA (2012)
2. Anastasakis, K., Bordbar, B., Küster., J.M.: Analysis of model transformations via Alloy. In: 4th Workshop on Model-Driven Engineering, Verification and Validation. pp. 47–56. Nashville, TN, USA (2007)
3. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: 13th International Conference on Model Driven Engineering Languages and Systems, pp. 121–135. Springer, Oslo, Norway (2010)
4. 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)
5. ATLAS Group: Specification of the ATL virtual machine. Tech. rep., Lina & INRIA Nantes (2005)
6. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: 4th International Conference on Formal Methods for Components and Objects, pp. 364–387. Springer, Amsterdam, Netherlands (2006)
7. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: 1st International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, pp. 49–69. Springer, Marseille, France (2005)
8. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM **53**(6), 139–143 (2010)
9. Benelellam, A., Gomez-Llana, A., Tisi, M., Cabot, J.: Distributed model-to-model transformation with ATL on MapReduce. In: 8th International Conference on Software Language Engineering, pp. 37–48. ACM, Pittsburg, USA (2015)
10. Berry, G.: Synchronous design and verification of critical embedded systems using SCADE and Esterel. In: 12th International Workshop on Formal Methods for Industrial Critical Systems, pp. 2–2. Springer, Berlin, Germany (2008)
11. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)
12. Bock, C., Cook, S., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., Tolbert, D.: OMG Unified Modeling Language (ver. 2.5). Tech. Rep. formal/2015-03-01 (2015)
13. Bornat, R.: Proving pointer programs in Hoare logic. In: International Conference on Mathematics of Program Construction, pp. 102–126. Springer, Ponte de Lima, Portugal (2000)
14. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Trans. Softw. Eng. **41**(5), 490–506 (2015)
15. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: 14th International Conference on Formal Engineering Methods, pp. 198–213. Springer, Kyoto, Japan (2012)
16. Calegari, D., Luna, C., Szasz, N., Tasistro, Á.: A type-theoretic framework for certified model transformations. In: 13th Brazilian Symposium on Formal Methods, pp. 112–127. Springer, Natal, Brazil (2011)
17. Calegari, D., Szasz, N.: Verification of model transformations: a survey of the state-of-the-art. Electron. Notes in Theor. Comput. Sci. **292**, 5–25 (2013)
18. Chan, K.: Formal proofs for QoS-oriented transformations. In: 10th International Conference Workshops on Enterprise Distributed Object Computing, pp. 41–41. IEEE, Hong Kong, China (2006)
19. Cheng, Z., Monahan, R., Power, J.F.: A sound execution semantics for ATL via translation validation. In: 8th International Conference on Model Transformation, pp. 133–148. Springer, L'Aquila, Italy (2015)
20. Cheng, Z., Monahan, R., Power, J.F.: Online repository for formalised EMFTVM bytecode language. https://github.com/veriatl/Compiler.Emftvm2Boogie (2016)
21. Cheng, Z.: Formal Verification of Relational Model Transformations Using an Intermediate Verification Language. Ph.D. thesis, Maynooth University (2016)
22. 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)
23. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, Los Angeles, California (1977)
24. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
25. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: 31st International Conference on Software Engineering, pp. 429–430. IEEE, Vancouver, British Columbia (2009)
26. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: 10th International Conference on Fundamental Approaches to Software Engineering, pp. 336–351. Springer, Braga, Portugal (2007)
27. Darvas, Á., Müller, P.: Reasoning about method calls in interface specifications. J. Object Technol. **5**(5), 59–85 (2006)
28. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Budapest, Hungary (2008)
29. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
30. Filliâtre, J.C., Paskevich, A.: Why3— where programs meet provers. In: 22nd European Symposium on Programming, pp. 125–128. Springer, Rome, Italy (2013)
31. Filliâtre, J.C.: Why: A multi-language multi-prover verification tool. Tech. rep., Université Paris Sud (2003)

32. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. Ann. Math. Artif. Intell. **55**(1–2), 101–122 (2009)
33. Guerra, E., de Lara, J.: Colouring: execution, debug and analysis of QVT-relations transformations through coloured Petri nets. Softw. Syst. Model. **13**(4), 1447–1472 (2014)
34. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
35. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, Cambridge (2004)
36. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: 14th International Conference on Model Driven Engineering Languages and Systems, pp. 653–667. Springer, Wellington, New Zealand (2011)
37. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
38. Jouault, F.: The resolve algorithm implemented in the ASM language. http://git.eclipse.org/c/mmt/org.eclipse.atl.git/tree/dsls/ATL/Compiler/ATL.acg (2007)
39. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
40. Klatt, B.: Xpand: a closer look at the model2text transformation language. http://bar54.de/benjamin.klatt-xpand.pdf (2007)
41. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman, Boston (2003)
42. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects Comput. **27**(1), 193–235 (2014)
43. Lehner, H., Müller, P.: Formal translation of bytecode into BoogiePL. In: 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation, pp. 35–50. Elsevier, Budapest, Hungary (2007)
44. Leino, K.R.M., Middelkoop, R.: Proving consistency of pure methods and model fields. In: 12th International Conference on Fundamental Approaches to Software Engineering, pp. 231–245. Springer, York, UK (2009)
45. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: 24th Annual ACM Symposium on Applied Computing, pp. 615–622. ACM, Hawaii, USA (2009)
46. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, pp. 348–370. Springer, Dakar, Senegal (2010)
47. Leino, K.R.M.: This is Boogie 2. http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf. Microsoft Research, Redmond, USA (2008)
48. Lúcio, L., Barroca, B., Amaral, V.: A technique for automatic validation of model transformations. In: 13th International Conference on Model Driven Engineering Languages and Systems, pp. 136–150. Springer, Oslo, Norway (2010)
49. Lúcio, L., Vangheluwe, H.: Model transformations to verify model transformations. In: 2nd Workshop on Verification of Model Transformations. Budapest, Hungary (2013)
50. Manna, Z., McCarthy, J.: Properties of programs and partial function logic. Mach. Intell. **5**, 27–38 (1969)
51. Mottu, J., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: 2nd European Conference on Model Driven Architecture-Foundations and Applications. pp. 376–390. Springer, Bilbao, Spain (2006)
52. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, pp. 151–166. Springer, London, UK (1998)
53. Poernomo, I., Terrell, J.: Correct-by-construction model transformations from partially ordered specifications in Coq. In: 12th International Conference on Formal Engineering Methods, pp. 56–73. Springer, Shanghai, China (2010)
54. Poernomo, I.: Proofs-as-model-transformations. In: 1st International Conference on Model Transformation, pp. 214–228. Springer, Zürich, Switzerland (2008)
55. Rahim, L.A., Whittle, J.: A survey of approaches for verifying model transformations. Softw. Syst. Model. **14**(2), 1003–1028 (2015)
56. Sahin, D., Kessentini, M., Wimmer, M., Deb, K.: Model transformation testing: a bi-level search-based software engineering approach. J. Softw. Evol. Process **27**(11), 821–837 (2015)
57. Schätz, B.: Verification of model transformations. In: 9th International Workshop on Graph Transformation and Visual Modeling Techniques, pp. 130–142. EASST, Paphos, Cyprus (2010)
58. Selim, G., Wang, S., Cordy, J., Dingel, J.: Model transformations for migrating legacy models: an industrial case study. In: 8th European Conference on Modelling Foundations and Applications, pp. 90–101. Springer, Lyngby, Denmark (2012)
59. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework, 2nd edn. Pearson Education, London (2008)
60. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Model. **12**(2), 387–414 (2013)
61. Tristan, J., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 295–305. ACM, San Jose, USA (2011)
62. Tristan, J., Leroy, X.: A simple, verified validator for software pipelining. In: 37th ACM Symposium on Principles of Programming Languages, pp. 83–92. ACM, Madrid, Spain (2010)
63. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. J. Object Technol. **10**(5), 1–29 (2011)
64. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Verifying Eiffel programs with Boogie. In: Computing Research Repository abs/1106.4700 (2011)
65. Varró, G., Varró, D., Friedl, K.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. In: 1st International Workshop on Graph and Model Transformations, pp. 191–205. Elsevier, Brighton, United Kingdom (2006)
66. Vépa, É., Bézivin, J., Brunelière, H., Jouault, F.: Measuring model repositories. In: Summary of the 2006 Model Size Metrics Workshop. Springer, Genoa, Italy (2006)
67. Vignaga, A.: Metrics for measuring ATL model transformations. Tech. rep., Universidad de Chile (2009)
68. Wagelaar, D., Iovino, L., Ruscio, D.D., Pierantonio, A.: Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: 5th International Conference on Model Transformation, pp. 192–207. Springer, Prague, Czech Republic (2012)
69. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: 14th International Conference on Model Driven Engineering Languages and Systems, pp. 623–637. Springer, Wellington, New Zealand (2011)
70. Wagelaar, D.: The resolve algorithm implemented in the EMFTVM language. http://git.eclipse.org/c/mmt/org.eclipse.atl.git/tree/plugins/org.eclipse.m2m.atl.emftvm/src/org/eclipse/m2m/atl/emftvm/util/OCLOperations.java (2011)
71. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)
72. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., Schwinger, W.: Right or wrong? Verification of model

transformations using colored Petri nets. In: 9th OOPSLA Workshop on Domain-Specific Modeling, pp. 101–106. Helsinki School of Economics, Orlando, USA (2009)
73. Wu, H., Monahan, R., Power, J.: Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In: 7th International Symposium on Theoretical Aspects of Software Engineering, pp. 175–182. IEEE, Birmingham, UK (2013)

Zheng Cheng [BSc, Beijing University of Civil Engineering and Architecture, 2007; MSc, Maynooth University, 2011; PhD., Maynooth University, 2016] is a postdoctoral researcher at AtlanMod Team (INRIA, Mines Nantes, LINA), France. His research interests lie in model-driven engineering, model transformation and deductive program verification.

Rosemary Monahan [BSc, University College Dublin, 1995; PhD., Dublin City University, 2010] is a lecturer in the Department of Computer Science at Maynooth University, Ireland, since 1999. Her research is concerned with the development of reliable software systems and is focused on the verification of object-oriented programs and the use of automatic verification tools.

James F. Power [BSc, University College Dublin, 1990; PhD., Dublin City University, 1995] is a lecturer in Computer Science at Maynooth University, Ireland. His research interests include program analysis and transformation, software verification and reverse engineering.