Towards a General Composition Semantics for Rule-Based Model Transformation

Dennis Wagelaar[1,⋆], Massimo Tisi[2], Jordi Cabot[2], and Frédéric Jouault[2]

[1] Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be
[2] École des Mines de Nantes, 4, rue Alfred Kastler, 44307 Nantes, France
{massimo.tisi,jordi.cabot,frederic.jouault}@inria.fr

Abstract. As model transformations have become an integral part of the automated software engineering lifecycle, reuse, modularisation, and composition of model transformations becomes important. One way to compose model transformations is to compose modules of transformation rules, and execute the composition as one transformation (internal composition). This kind of composition can provide fine-grained semantics, as it is part of the transformation language. This paper aims to generalise two internal composition mechanisms for rule-based transformation languages, module import and rule inheritance, by providing executable semantics for the composition mechanisms within a virtual machine. The generality of the virtual machine is demonstrated for different rule-based transformation languages by compiling those languages to, and executing them on this virtual machine. We will discuss how ATL and graph transformations can be mapped to modules and rules inside the virtual machine.

Keywords: Model transformation, Model transformation composition, ATL, Graph transformation.

1 Introduction

Model transformations play a central role in MDE, and have become an integral part of the automated software engineering lifecycle, just like build script interpreters and compilers. In order to keep this automated lifecycle maintainable, model transformations will have to be reusable, modular, and composable. We can distinguish between two kinds of composition for model transformation: *external* composition and *internal* composition [1]. External composition refers to a chain of several model transformation executions, where models are passed from one transformation to another. Internal composition refers to the composition of multiple transformation rules and/or modules into one transformation module, which can then be executed as a whole.

⋆ The author's work is funded by a postdoctoral research grant provided by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders).

J. Whittle, T. Clark, and T. Kühne (Eds.): MODELS 2011, LNCS 6981, pp. 623–637, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The advantage of external composition is its independence of the transformation language, while internal composition relies on specific transformation language semantics and/or constructs (e.g. modules, rules, operations, etc.). It therefore often applies to one transformation language only, as language semantics generally apply to one language only. The advantage of internal composition is the richer, more fine-grained composition semantics it can provide. It is possible to refine or redefine existing rules, add new rules, etc., as long as there is a common notion of what a rule is.

Different transformation languages have different strengths, which has been demonstrated by the Transformation Tool Contest workshop series[1]. The ability to perform fine-grained composition of transformation rules expressed in different languages is a powerful tool for tackling complex transformation problems, as each language can be used for their strong points.

This paper aims to mitigate the problem of internal composition being specific to one transformation language by defining the composition mechanism within the context of a transformation virtual machine (VM). The VM provides a common, executable semantics for (composition of) transformation modules and rules. Two internal composition mechanisms for rule-based transformation languages are generalised in this way: module import and rule inheritance. The VM, called EMF Transformation Virtual Machine (EMFTVM), is based on the Eclipse Modeling Framework (EMF) [2], which represents a de facto standard for modelling today. As a result, the proposed composition mechanisms are specific to EMF.

The generality of EMFTVM – within the scope of EMF – is demonstrated by compiling more than one rule-based model transformation language to the VM, and by extension provide executable semantics for those languages. As a proof of concept, we discuss how ATL [3] and graph transformations [4] can be mapped to modules and rules in our VM. For this purpose, we've developed SimpleGT, a minimal graph transformation language on top of EMF, based on double push-out (DPO) semantics. The combination of ATL and SimpleGT already provides a non-trivial spectrum of rule-based languages, as ATL is a model *mapping* language, and SimpleGT is a recursive model *rewriting* language. This difference is discussed in detail in the paper.

The generality of EMFTVM also applies to the composition mechanisms implemented in EMFTVM: ATL's and SimpleGT's notion of module import and rule inheritance are mapped to the same implementation, and therefore have common executable semantics. In the long term, EMFTVM may evolve towards a general interoperability solution for model transformation languages that leverages commonalities between languages.

The rest of this paper is organised as follows: in section 2, we discuss related work. In section 3, we briefly explain the EMFTVM language. Then, we discuss how the VM implements rule inheritance in section 4, and module import in section 5. Section 6 discusses how ATL and SimpleGT are mapped to modules and rules in our VM. Section 7 concludes this paper.

[1] http://planet-research20.org/ttc2011/

2 Related Work

2.1 Common Semantics and Virtual Machines

In the domain of model transformation, there have been two efforts to provide common executable semantics for multiple transformation languages. One of these concerns the alignment of ATL and QVT Operational [5]. The executable semantics are provided by the ATL VM in this case. Another such effort is the ATC VM[2], which aims to provide a common execution framework for languages such as QVT or RubyTL. In both cases, composition possibilities are limited, because rules are compiled away into low-level primitives. The necessary meta-data to perform rule composition, such as what code belongs to what rule, what are the rule's input/output elements, and what are a rule's super-rules, are no longer available.

2.2 Rule Inheritance

Rule inheritance allows a transformation rule to specify one or more super-rules, where structure and behaviour of super-rules is inherited cf. object-oriented inheritance. According to [6], there are currently three model transformation languages that include an explicit notion of rule inheritance: ATL [3], the Epsilon Transformation Language (ETL) [7], and Triple Graph Grammars (TGG) [8]. Each of these languages assumes slightly different semantics for rule inheritance, and conflict with each other at specific points. For example, ETL triggers a super-rule whenever its sub-rule triggers , whereas ATL will only trigger a sub-rule if its super-rule triggers first. TGG in turn requires you to include the entire super-rule as part of each sub-rule, which allows both ETL's and ATL's rule inheritance strategy to be used.

QVT Operational and Relations [9] include "when" and "where" clauses, which allow for triggering other mappings/relations from the context of a mapping/relation. A "when" clause requires the referenced mapping/relation to match first, before the current mapping/relation is applied. This corresponds to the rule inheritance strategy for ATL. A "where" clause enforces the referenced mapping/relation to be applied before the current mapping/relation is applied. This corresponds to the rule inheritance strategy for ETL.

The VIATRA2 language [10] uses reusable patterns to specify rule trigger conditions. Rules can refer to patterns, and patterns may include other patterns. This results in a kind of "inheritance hierarchy" of patterns, where each pattern requires all its included patterns to match first. VIATRA2 also uses the pattern hierarchy to perform optimised matching [11].

2.3 Module Import

Module import allows model transformation languages to separate transformation rules into multiple modules, and allow a module to include the contents

[2] `http://sourceforge.net/projects/atc/`

of one or more other modules. ATL provides a feature called module superimposition [12], which allows for combining multiple transformation modules by loading them on top of each other, redefining rules and helpers with the same signature. ETL supports a built-in module import construct, which loads other modules during the loading of the current module. Elements with the same signature are also redefined in ETL. QVT Operational uses “access” and “extends” to compose modules. “Access” loads another module in its own namespace, and all its mappings must be explicitly triggered. “Extends” loads another module into the current namespace, where the current module redefines any mappings with the same signature in the extended module. VIATRA2 supports a module import construct as well, which enables fine-grained reuse of patterns. It is unclear whether VIATRA2’s module import also supports redefinition.

3 Transformation Virtual Machine Language

The EMFTVM is a stack-based VM (i.e. instructions communicate values via a stack), and uses a low-level bytecode language to describe model transformations. The main feature of this bytecode language is that it includes an explicit representation of transformation *modules* and *rules*. This decision allows performing module and rule composition on the bytecode itself, as all necessary meta-data is available as a first-class entity in the bytecode. This section discusses the two main EMFTVM bytecode language features that are relevant for module import and rule inheritance: modules and rules.

3.1 Modules

EMFTVM bytecode is organised into *modules*, which represent self-contained units of execution. Each module consists of a number of *fields*, *operations*, and *rules*. Fields and operations can be static or dynamic, similar to Java fields and methods. Modules may *import* other modules, as is further explained in section 5.

Instructions are organised into *code blocks*. Fig. 1 shows the structure of code blocks. Code blocks are executable lists of instructions, and have a number of local variables and a local stack space. Code blocks are used to represent operation bodies and field initialisers. Code blocks may also have nested code blocks, which effectively represent *closures*. Closures are nameless functions that can be invoked or passed as parameters to other functions. Closures are helpful for the implementation of OCL’s higher-order operations, such as `select` and `collect`. Closures are also helpful to simplify compilation of source transformation languages, as each source language AST node can be locally compiled into its own code block, and may be nested into the correct place. Such closures may be inlined after compilation.

EMFTVM supports 47 different instructions[3]. Apart from the general-purpose instructions for control flow, several EMF-specific instructions exist, such as

[3] `http://soft.vub.ac.be/viewvc/*checkout*/EMFTVM/trunk/emftvm/EMFTVM.html`

SET, GET, ADD, REMOVE, and INSERT. While *mapping* style transformation languages typically SET element properties, *rewriting* style languages typically ADD and REMOVE element properties. As EMF properties are ordered lists, an INSERT instruction allows one to insert a property value at a specific index.

Finally, modules specify a number of *input*, *inout*, and *output* models. This distinction allows one to enforce read-only or write-only constraints at run-time: input models are read-only, output models write-only, and inout models can be read and written.

3.2 Rules

Fig. 1 shows the part of the EMFTVM metamodel that defines rules and code blocks. Rules consist of input elements, output elements, a *matcher* code block, *applier* code block, and *post-apply* code block. This distinction between *matcher*, *applier*, and *post-apply* allows one to execute rules in stages: the *matcher* filters potential input element matches, the *applier* assigns element properties and deletes elements, and the *post-apply* block contains code that should be run after a rule has been applied. EMFTVM provides a framework for automatic matching and tracing, which invokes these three different code blocks at specific stages.

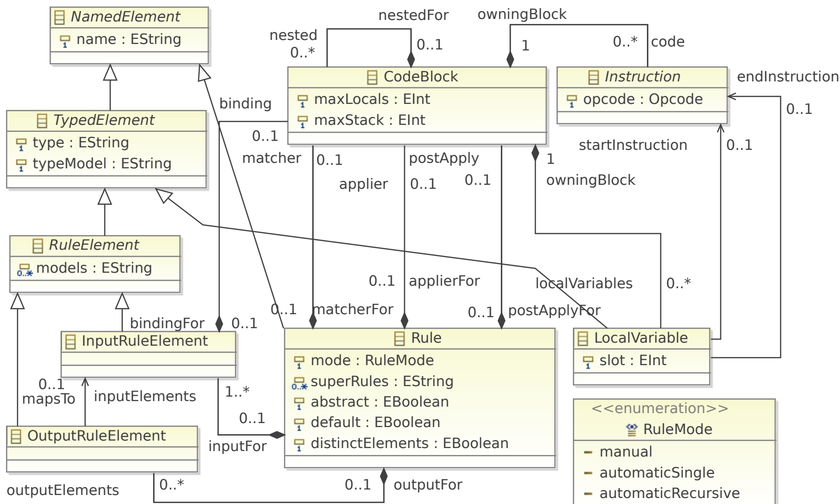

Fig. 1. Structure of EMFTVM rules and code blocks

Input elements can have a *binding* code block. This allows EMFTVM to apply a *search plan* strategy [10] in its automatic matcher. Each *binding* block calculates the valid values for an input element, given the values of the input elements that have already been bound (either by iteration or by another *binding*).

Furthermore, rules have a name that is unique within its module, and can have a number of super-rules. These super-rules are stored as names only, and are resolved at load-time, when rules are composed. This is done to facilitate interaction with the module import mechanism, and is further discussed in section 5. Super-rules and rule inheritance are further explained in section 4.

Rules can be *abstract*, which means that they are only applied in combination with a non-abstract sub-rule. A rule may create *default* traces, which allows the transformation module to *resolve* target elements from a (list of) source element(s). Default traces have as consequence that the same input pattern may not be matched by another rule that creates default traces, as this would result in ambiguous source-target value resolution. Rules may also match against *distinct elements*, which means that no two elements in a single input pattern match can be equal.

Finally, rules have an execution *mode*, which can be either *manual*, *automatic single*, or *automatic recursive*. *Manual* rules have to be explicitly invoked. *Automatic single* rules are matched once, then applied once by the automatic matching framework. *Automatic recursive* rules are matched and applied by the automatic matching framework until there are no more matches.

The next section proposes a common semantics for rule inheritance.

4 Rule Inheritance

Rule inheritance in EMFTVM allows rules to specify a list of super-rules, whereby sub-rules can only match on input that has also matched against their super-rules. As a result, rule inheritance serves as an optimisation strategy that only tries to match sub-rules whenever their super-rules have already matched. This effectively represents a RETE network, such as applied in VIATRA2 [11]. Rule inheritance also serves as a reuse mechanism, whereby sub-rules can reuse and extend the input pattern and output pattern with new elements. Reducing the number of input elements – or output elements – is not possible, and any omitted input/output elements are implicitly inherited from the super-rule. However, super-rule input/output elements must be repeated in the sub-rule in case lexical access to the elements is required (e.g. in the applier or post-apply block).

The EMFTVM rule inheritance mechanism supports multiple inheritance, which requires all super-rules to have matched on the same input before trying to match the sub-rule. Before applying the sub-rule, all super-rules are applied in the order they are specified in the sub-rule. Fig. 2 outlines the semantics for rule matching in the context of rule inheritance. Each rule is represented by a box with compartments. The left compartment contains the input elements, whereas the right compartment contains the output elements. Each input/output element is specified by a label and a type (i.e. `label:Type`).

Rule `R3` in the figure only matches against input elements that have also been matched by super-rules `R1` and `R2`. Input/output elements correspond by label: input element `b:B` in rule `R1`, and `b:D` in rule `R2` are the same as input element `b:F` in rule `R3` for any match of rule `R3`. Therefore, `R3` only matches `b`'s that are an instance of `B`, `D`, and `F`.

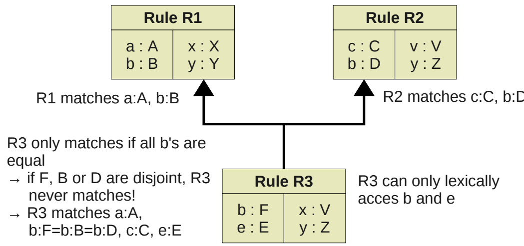

Fig. 2. Matching semantics for rule inheritance

As the number of input/output elements cannot be reduced in sub-rules, `R3` is considered to inherit the input elements `a:A` and `c:C` from rules `R1` and `R2`, respectively, and output element `v:V` from rule `R2`. Rule `R3` cannot lexically access those elements, however, as the EMFTVM engine does not pass them as parameters to `R3`'s matcher, applier, and post-apply code blocks.

It is only possible to define super-rule relations between rules of the same kind: manual, automatic, or recursively automatic, and default or non-default. This is because super- and sub-rules are executed together according to the same execution semantics. Taking this into account, the matching semantics of each rule remains sound, even if any of the rules is replaced by an arbitrary other rule (of the same kind). If rules are truly incompatible, they will simply not produce any combined match.

Fig. 3 outlines the semantics for rule application in the context of rule inheritance. Whereas the matching semantics are sound for any change in the rule hierarchy, the application semantics comes with some type safety constraints. The types of all input elements are already guaranteed by the matching algorithm (matches only occur on the specified types). However, the types of the output elements must be compatible between super- and sub-rule. The rule application algorithm creates output elements that are instances of the types specified in the sub-rule. Therefore, those types must be *co-variant* with the types specified for the same elements in the super-rule. For example: an element `x : V` is created for each match of `R3`, but is considered as `x : X` in the application of `R1`. Therefore, `V` must be *co-variant* with `X`: each instance of `V` must also be an instance of `X`. Similarly, for the creation of `y : Z` for `R3`, and `y : Y` in `R1`, `Z` must be co-variant with `Y`. These type safety constraints may be checked at load-time by the virtual machine.

The automatic rule matching framework performs optimised matching of rule hierarchies, while being implemented reflectively, i.e. looking up super-rules and input/output elements and their types at run-time. The algorithm is split up into two phases: (1) matching the single automatic rules and (2) matching the recursive automatic rules. The algorithm for single rules works as follows:

1. All rules without super-rules are matched, and their matches (tuple of input elements) are stored.

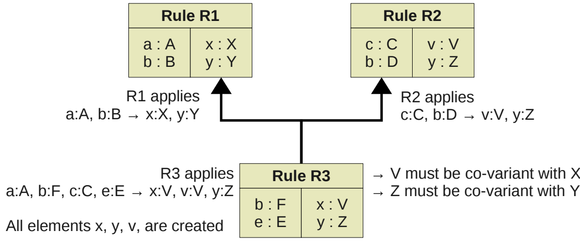

Fig. 3. Application semantics for rule inheritance

2. All rules for which all of their super-rules have matched the same elements are now matched on those elements, and their matches are stored. For all matches, the super-rule matches are removed.
3. The previous step is repeated until all applicable rules are processed.
4. For all matches of non-abstract rules, output elements are created, and the match tuple is converted to a trace tuple that includes the output elements.
5. For all traces, the corresponding rule applier code block is invoked, super-rules first, then the sub-rule.
6. For all traces, the corresponding rule post-apply code block is invoked, super-rules first, then the sub-rule.

Single automatic rules are expected to match on elements from a different model (e.g. an *input* model) than the model in which the rules are applied (e.g. an *output* model). This guarantees that previously found matches are not invalidated by applying rules.

The same cannot be expected for recursive rules, which must be able to match on their own output. Therefore, recursive rules can match on elements from any model (e.g. *inout* models). The algorithm for recursive rules takes this into account, by re-matching after each apply:

1. Rules without super-rules are matched first. For rules with sub-rules, all matches (tuple of input elements) are stored, while only the first match is stored for rules without sub-rules. If a (non-abstract) rule without sub-rules matches, it is applied[4], the recorded matches are cleared, and the algorithm restarts.
2. Rules for which all of their super-rules have matched the same elements are now matched on those elements. Again, for rules with sub-rules, all matches are stored, while only the first match is stored for rules without sub-rules. For all matches, the super-rule matches are removed. If a (non-abstract) rule without sub-rules matches, it is applied, the recorded matches are cleared, and the algorithm restarts.

[4] For recursive rules, applying involves converting a match to a trace, creating output elements, and invoking the applier block and post-apply block.

3. When all rules have been processed, and (non-abstract) matches have been recorded, the first of those matches is applied, the recorded matches are cleared, and the algorithm restarts. Otherwise, the algorithm ends.

These algorithms ensure that sub-rules are only matched for the elements that have already been matched by their super-rules, with no unnecessary matching. They also ensure that sub-rules cannot widen the initial input element type constraints and constraints encoded in the matcher code block of the super-rules.

When executing single and recursive automatic rules together, they may operate on the same models in the following way: the single rules transform from an *input* model to an *inout* model, and the recursive rules then further transform the *inout* model.

Even though the different kinds of rules use different matching algorithms, these algorithms share their implementation of rule inheritance. A unified semantics for rule inheritance is enforced in this way.

The following section proposes a common semantics for module import.

5 Module Import

EMFTVM supports module import via the "imports" attribute of each module, which lists a number of module names. These names are resolved at load-time by the VM. Fig. 4 shows how EMFTVM module import works. Each module loads its imported modules before loading itself, in the specified order. For example, module `M1` requires that first `M2` is loaded, and then `M3`. The first step is then to start loading `M2` (1). Then, `M2` requires that `M4` and `M3` are loaded before itself. Therefore, `M4` is loaded (2), and then `M3` is loaded (3), which finds that its imported `M4` was already loaded (4). Now, `M2` can be loaded, and `M1` finds that `M3` was already loaded (5). Finally, `M1` is loaded. Circular imports – and self-imports – are ignored.

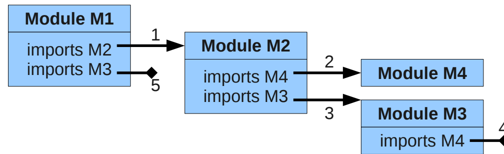

Fig. 4. Module import semantics

Module import supports *redefinition* of fields, operations, and rules that were already specified in an imported module. Whenever a module is imported, its fields, operations, and rules are registered in the VM's lookup table: rules are registered by name, whereas fields and operations are also registered by their context and parameter types[5]. Fields and operations are only *redefined* by an

[5] Static fields/operations have a separate lookup table.

importing module if the context and parameter types match. Fields and operations with different context/parameter types are *overloaded* instead. Hence, redefinition of fields and operations is always type-safe. Redefined elements are completely removed, and cannot be accessed by the redefining element.

Because rules are only registered by name, any rule with the same name may redefine an existing rule. That means additional constraint checking is required for rule redefinition. Rules must be of the same kind – manual/automatic single/automatic recursive, and default/non-default – to allow sound redefinition. After all modules are imported, and all rule redefinition has been performed, the super-rules for each rule are resolved. At this time, the type safety checks for rule inheritance are performed (see section 4).

Finally, in case of conflicting specified importing orders, the depth-first loading order, as shown in Fig. 4, is followed. For example, if `M1` specified another `imports M4` statement after `imports M3`, the loading algorithm would still load `M3` after `M4`. This is considered correct, because by specifying `imports M4`, `M3` states that it wants the opportunity to redefine elements of `M4`. `M1` may still redefine all elements, as it is the last module to be loaded.

Module import is considered transitive: if `M1` imports `M2`, and `M2` imports `M4`, then `M1` imports `M4`, and can redefine elements of `M4`.

6 Mapping of Rule-Based Model Transformation Languages

To demonstrate the generality of the previously explained composition mechanisms, a mapping from ATL and SimpleGT to the EMFTVM is presented. ATL is an established, *mapping*-style model transformation language, and SimpleGT is a proof-of-concept, *rewriting*-style model transformation language, based on double push-out (DPO) graph transformation semantics. By mapping these two different languages to the same VM, we effectively provide common executable semantics for both languages, including a common semantics for the composition mechanisms discussed before.

6.1 ATL

ATL transformation definitions consist of modules, which can contain different kinds of rules, helper attributes, and helper methods. The mapping of ATL to EMFTVM is straightforward for the most part: Table 1 provides an overview of how ATL constructs are mapped to EMFTVM constructs.

As ATL includes OCL to do its model navigation, OCL support also has to be included in the mapping. EMFTVM forms a symbiosis with the underlying Java run-time environment, and allows the lookup of Java types and invocation of Java methods. OCL support is provided in the form of a natively implemented EMFTVM module of operations, and a set of natively implemented collection types (Sequence, Set, Bag, OrderedSet). Higher-order collection operations, such as `select` and `collect`, take an EMFTVM code block as an argument. The

Table 1. Mapping of ATL constructs to EMFTVM constructs

ATL construct → EMFTVM construct
module → module
uses → imports
input model → input model
output model → output model
metamodel → metamodel
matched rule → automatic single default rule
nodefault matched rule → automatic single rule
lazy rule → manual rule
unique lazy rule → manual default rule
rule input element → input element
rule output element → output element
input pattern filter expression → code in rule matcher block
output pattern bindings → code in rule applier block
code in "**do**" block → code in rule post-apply block
matched rule variables → rule fields
called rule → static operation
called rule variables → local variables in operation code block
entrypoint rule → static operation, called from `main`
endpoint rule → static operation, called from `main`
helper attribute without context → static field
helper attribute with context → field
helper method without context → static operation
helper method with context → operation

complete mapping of ATL to EMFTVM is described in the ATL-to-EMFTVM compiler[6]. This compiler is written in ATL, and compiled by itself to EMFTVM.

6.2 Graph Transformations

Most existing graph transformation languages have already evolved into a fairly complex language (e.g. using implicit NAC expressions [13] and control flow constructs [14]). However, none of them support rule inheritance yet[7], and the transformation language has to be altered to support it. Therefore, we introduce a basic, proof-of-concept graph transformation language with built-in rule inheritance and module import support: SimpleGT. SimpleGT is a textual graph rewriting language, based on *double push-out semantics* (DPO): rules include an input graph, correspondence graph, and output graph, where the input graph is deleted, the correspondence graph is left unchanged, and the output graph is created. The correspondence graph is implicit, and is represented by the

[6] `http://tinyurl.com/ATLtoEMFTVM-atl`

[7] Triple Graph Grammars (TGG) do support rule inheritance, but form a different class of graph transformation.

```
module InlineCodeblocks;
transform M : EMFTVM;
rule RetargetInvoke_cbLocalVariableStart { ... }
rule RetargetInvoke_cbLocalVariableEnd {
    from lv : EMFTVM!LocalVariable (endInstruction =~ invoke_cb),
         invoke_cb : EMFTVM!Invoke_cb (codeBlock =~ nestedCb),
         nestedCb : EMFTVM!CodeBlock (code =~| last),
         last : EMFTVM!Instruction
    to lv : EMFTVM!LocalVariable (endInstruction =~ last),
         invoke_cb : EMFTVM!Invoke_cb (codeBlock =~ nestedCb),
         nestedCb : EMFTVM!CodeBlock (code =~ last),
         last : EMFTVM!Instruction }
abstract rule Invoke_cb {
    from cb : EMFTVM!CodeBlock (code =~ invoke_cb),
         invoke_cb : EMFTVM!Invoke_cb
    to cb : EMFTVM!CodeBlock (code =~ invoke_cb),
       invoke_cb : EMFTVM!Invoke_cb }
rule Invoke_cb_inline_locals extends Invoke_cb {
    from cb : EMFTVM!CodeBlock (code =~ invoke_cb),
         invoke_cb : EMFTVM!Invoke_cb (codeBlock =~ nestedCb),
         nestedCb : EMFTVM!CodeBlock (localVariables =~ lv),
         lv : EMFTVM!LocalVariable (slot =~ lv.slot)
    to cb : EMFTVM!CodeBlock (code =~ invoke_cb,
                localVariables =~ lv),
       invoke_cb : EMFTVM!Invoke_cb (codeBlock =~ nestedCb),
       nestedCb : EMFTVM!CodeBlock,
       lv : EMFTVM!LocalVariable }
rule Invoke_cb_inline extends Invoke_cb {
    from cb : EMFTVM!CodeBlock (code =~ invoke_cb),
         invoke_cb : EMFTVM!Invoke_cb (codeBlock =~ nestedCb),
         nestedCb : EMFTVM!CodeBlock
    not nestedCb : EMFTVM!CodeBlock (localVariables =~ lv),
         lv : EMFTVM!LocalVariable
    to cb : EMFTVM!CodeBlock (code =~ invoke_cb,
         code =~ nestedCb.code before invoke_cb,
         lineNumbers =~ nestedCb.lineNumbers,
         nested =~ nestedCb.nested before nestedCb),
       invoke_cb : EMFTVM!Invoke_cb }
```

Listing 1.1. Excerpt of InlineCodeblocks SimpleGT module

intersection of the input and output graph. SimpleGT uses explicit *negative application condition* graphs (NACs), which specify input patterns that prevent the rule from matching.

Listing 1.1 shows an excerpt of a SimpleGT transformation module, named "InlineCodeblocks". This transformation rewrites INVOKE_CB instructions by inlining the invoked code block into the calling code block. SimpleGT uses `from` to specify the input pattern, and `to` to specify the output pattern. The common elements in the input and output pattern form the correspondence graph, and is not altered by EMFTVM. Nodes map to EMF EObjects and edges map to EMF EReferences. A node is specified using a label and type. An edge is specified using the '`=~`' matching operator: this operator specifies the existence of an edge (or EReference value). In addition, the '`=~`' operator can be used to match node attribute values (EAttributes).

The rules `RetargetInvoke_cbLocalVariableStart` and `RetargetInvoke_cb-LocalVariableEnd` re-map the start and end instruction of local variables that refer to INVOKE_CB instructions. Only the latter rule is listed here, as it

Table 2. Mapping of SimpleGT constructs to EMFTVM constructs

SimpleGT construct →	EMFTVM construct
module →	module
imports →	imports
model →	inout model
metamodel →	metamodel
rule →	automatic recursive distinct rule
input nodes →	input element
nac nodes →	*code in rule matcher block*
output nodes →	output element *if new element*
unchanged edges →	code in rule matcher block
deleted edges →	code in rule matcher block and applier block
new edges →	code in rule applier block
deleted nodes →	code in rule matcher block and applier block

includes a special feature: EMF models have *ordered* edges; the '`=~|`' operator allows one to match the *last* edge going out from a node (the regular '`=~`' operator always matches the *first* edge). In this case, the `endInstruction` of local variable `lv` should be re-mapped to the last instruction in the nested code block.

The `Invoke_cb` rule is an example of an *abstract* rule that is inherited by `Invoke_cb_inline_locals` and `Invoke_cb_inline`. An abstract rule is only applied when a non-abstract sub-rule is applied. Conversely, the sub-rules only match when all super-rules have matched. `Invoke_cb_inline_locals` moves one local variable at a time into the calling code block, while re-setting the assigned local variable slot (the EMFTVM metamodel implementation automatically sets this again on read access). `Invoke_cb_inline` performs the actual inlining, and moves the code (i.e. instructions), nested code blocks, and line number mappings from each invoked code block into its calling code block. The `before` keyword is used to enforce *insert* semantics instead of *append* semantics (the default): the code of the nested code block should be inserted *before* the subject INVOKE_CB instruction.

The remainder of the transformation module[8] is omitted, as it does not introduce new SimpleGT constructs. An overview of the mapping of SimpleGT to EMFTVM is provided in Table 2. The complete mapping of SimpleGT to EMFTVM is described in the SimpleGT-to-EMFTVM compiler[9], which is written in ATL, and compiled to/executed in EMFTVM.

SimpleGT rules map to automatic, recursive, non-default rules in EMFTVM. Input nodes map to input elements, output nodes map to output elements *only if* they did not occur in the input pattern. NAC nodes are not explicitly represented by rule elements in EMFTVM: the goal is to *not* match them. Instead, they are represented in the rule matcher code block, to make sure they do not occur as

[8] `http://tinyurl.com/InlineCodeblocks-simplegt`
[9] `http://tinyurl.com/SimpleGTtoEMFTVM-atl`

part of the input graph. The *binding* code block of EMFTVM rule input elements is used to implement a search plan strategy, where input node values are derived from other input node values. The search plan code for NAC nodes is embedded in the rule matcher code block.

7 Conclusion and Future Work

This paper has presented an approach to achieve a general semantics for two internal composition mechanisms for rule-based model transformation languages: module import and rule inheritance. These general semantics are achieved in three steps: (1) module import and rule inheritance are defined within a virtual machine (VM) for model transformation, named EMFTVM, (2) the generality of the VM is demonstrated by translating two distinct transformation languages, ATL and graph transformations, to the VM, and (3) by translating ATL and graph transformations to the same VM, a common semantics for module import and rule inheritance applies to those languages.

The generality of the presented semantics is limited by two factors: (1) EMFTVM is specific to EMF models, and (2) only two rule-based languages have been translated to EMFTVM. As EMF is a de facto standard for modelling, and many transformation languages target EMF [3,7,9,10,14,15,16], the scope of EMF is considered sufficiently relevant to the modelling community. The fact that only ATL and SimpleGT, a proof-of-concept graph transformation language, have been translated to EMFTVM is mitigated by the nature of both languages. ATL is a model *mapping* language, which uses a single rule matching phase, after which all rules are applied. SimpleGT is a recursive model *rewriting* language, which applies its rules recursively until no more matches can be found. Both are very different in rule matching and application semantics, but are still able to share the semantics for module import and rule inheritance. Any languages with semantics similar to either ATL (i.e. the QVT-like languages) or SimpleGT (i.e. graph transformation languages) can likely be mapped to EMFTVM as well.

As EMFTVM implements the entire ATL and SimpleGT languages, it provides a complete interoperability solution for these languages (including ATL's rule invocation and implicit tracing mechanism). Over time, EMFTVM may evolve as a general interoperability solution, as more languages are mapped to it. It is currently not possible to map *synchronisation*-style languages, such as QVT Relations [9] and Triple Graph Grammars (TGG) [8], to EMFTVM. These languages try to first match output elements, and will create them if not found. Current EMFTVM output elements are always created.

References

1. Kleppe, A.G.: First European Workshop on Composition of Model Transformations - CMT 2006. Technical Report TR-CTIT-06-34, Enschede (2006)
2. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse Series. Addison Wesley Professional, Reading (2003)

3. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation, 1st edn. Springer, Heidelberg (2006)
5. Jouault, F., Kurtev, I.: On the Architectural Alignment of ATL and QVT. In: Proceedings SAC 2006 (2006)
6. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 31–46. Springer, Heidelberg (2011)
7. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
8. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
9. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification, Final Adopted Specification, ptc/05-11-01 (2005)
10. Varró, G., Friedl, K., Varró, D.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. Electr. Notes Theor. Comput. Sci. 152, 191–205 (2006)
11. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: Proceedings of GRaMoT 2008, pp. 25–32. ACM Press, New York (2008)
12. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software and Systems Modeling 9, 285–309 (2009)
13. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
14. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
15. Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
16. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 62–76. Springer, Heidelberg (2005)