A Revised Semantics for Rule Inheritance and Module Superimposition in ATL

Dennis Wagelaar*

Software Languages Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be, http://soft.vub.ac.be

Abstract. There are two composition mechanisms in ATL that have evolved separately, and therefore are not well-aligned. These are rule inheritance and module superimposition. Both of these composition mechanisms had different goals in mind: rule inheritance aims to increase reuse at the rule level, as well as optimise the performance of the matching process, while module superimposition aims to increase reuse at the module level, and allows for incremental compilation whenever a single module changes. To achieve these goals, rule inheritance was in-lined at compile-time, while superimposed modules were composed at load-time. As a result, rule inheritance is limited to single modules, while module superimposition rule overriding does not work well on rule inheritance hierarchies. This paper aims to reconcile the two composition mechanisms by defining both at load-time/run-time, while respecting the original goals. In addition, rule inheritance is extended from single to multiple inheritance.

Keywords: Model transformation, ATL, rule inheritance, module superimposition

1 Introduction

Over time, a number of composition mechanisms have been developed for ATL: implicit tracing between matched rules [7], invocation of called/lazy rules, rule inheritance [2], and module superimposition [5]. While most of these work well together, and are well-defined in combination, two composition mechanisms are in conflict with each other. These are rule inheritance and module superimposition. Both have evolved separately, and with different goals in mind. Rule inheritance aims to facilitate reuse of (parts of) transformation rules, as well as optimise the performance of the matching process. Module superimposition aims to facilitate reuse of (parts of) transformation modules, as well as improving scalability of ATL development by enabling incremental module compilation whenever one of the modules in a composition changes.

* 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)

ATL's rule inheritance works by in-lining rule inheritance hierarchies at compile-time. An optimised matcher operation first matches the super-rules, and then matches the sub-rules on the input that was matched by the super-rule. Module superimposition enables incremental module compilation by performing the module composition on the bytecode at load-time, i.e. after the modules have been compiled. As a result, rule inheritance cannot work outside the boundaries of a single module, while module superimposition does not work well on rule inheritance hierarchies, due to multiple rules being represented in a single matcher operation.

One situation in which it is useful to combine module superimposition with rule inheritance, is to organise rule inheritance hierarchies over multiple modules, allowing the reuse of super-rules across modules. Another situation is the combination of superimposition's rule redefinition with rule inheritance: by redefining a super-rule, the behaviour of all sub-rules can be modified centrally; it is normally not possible to "inject" new behaviour in existing rule hierarchies, but one can only extend/refine existing rules.

This paper aims to reconcile the two composition mechanisms by defining both at the same stage, i.e. load-time/run-time, and on the same artefact: bytecode. At the same time, the original goals of rule inheritance and module superimposition are respected and upheld. For this purpose, a new virtual machine and bytecode format have been developed: the EMF Transformation Virtual Machine (EMFTVM). Finally, the semantics of ATL's rule inheritance are extended to support multiple inheritance.

The rest of this paper is structured as follows: section 2 discusses related work. Section 3 discusses the architecture and bytecode format of the EMFTVM. Section 4 explains the new semantics for rule inheritance, while section 5 explains the new semantics for module superimposition. Finally, section 7 concludes this paper.

2 Related work

This paper discusses a revised semantics for rule inheritance and module superimposition – which is a form of module composition – in ATL. These semantics are defined in a new transformation virtual machine. In this section, we first discuss related work in the domains of rule inheritance and module composition. Then, we discuss other transformation virtual machines.

2.1 Rule inheritance

According to [6], there are currently three known model transformation languages that support rule inheritance: ATL, the Epsilon Transformation Language (ETL) [1], and Triple Graph Grammars (TGG) [4]. In the comparison done by Wimmer at al. in [6], several shortcomings as well as advantages of ATL's rule inheritance mechanism were brought forward. We will focus on the issues that we tackle in this paper:

No multiple inheritance ATL rules do not support multiple inheritance. Both ETL and TGG support multiple inheritance. As EMF metamodels allow for multiple inheritance, it seems appropriate that any transformation language operating on EMF models follows this decision.

No extension of input pattern ATL does not allow for adding input elements in sub-rules. TGG has no problems adding new input elements in sub-rules. ETL rules always have exactly one input element.

No reduction of input pattern ATL does not allow for removing input elements in sub-rules. Also, neither TGG nor ETL support this. This is because the super-rule cannot be applied safely, once expected input elements are taken out.

No conflict detection between sibling sub-rules ATL normally detects when model elements match against more than one rule, and gives an error message when that happens. For sub-rules that inherit from the same super-rule, conflicts are not detected, and only the first sub-rule matches.

2.2 Module composition

In [5], we've compared a number of alternative transformation module composition mechanisms. In this paper, we will also add a brief discussion of VIATRA2's composition mechanism. Table 1 provides an updated comparison.

The rule-based model transformation languages in Table 1 that have a module concept – ETL, QVT, RubyTL, VIATRA2 – also have a form of module composition. The remaining languages focus on rule composition. If module composition is supported, it is typically an "import" style composition mechanism, where each module declares a number of imported modules. In the case of ETL and QVT, rule redefinition is also supported.

ATL's module superimposition differs in that respect: even though ATL has a **uses** clause, where imported modules/libraries can be declared, the run-time parameters (i.e. "launch configuration") decide which modules are loaded, and in which order. Even though it is considered good practice to declare any superimposed-upon modules in the **uses** clause, this is not enforced. As a consequence, inconsistencies may arise between the **uses** declaration and the run-time parameters for a given transformation module. The ability to redefine the superimposition order for a given set of modules is of limited value: modules are typically designed to be superimposed on specific other modules. For these reasons, it makes sense to re-align ATL's module superimposition semantics with the more common "import" style semantics found in other languages.

2.3 Transformation virtual machines

Apart from ATL, there are two more virtual machines for model transformation (based on EMF): the Atomic Transformation Code (ATC) VM [3], and IDC[1]. ATC aims to provide a low-level language for the implementation of QVT. It is

[1] http://modelum.es/trac/eclectic/

Language	Composition mechanisms	Key advantage/disadvantage
Graph transformation	Sequencing	In-place transformation allows free rule interaction/ Rule interaction easily becomes complex and can generate conflicts
ETL	Strategies, module import, workflows	Reuse as well as replace existing rules/ Tight coupling and possible overriding conflicts
QVT OM	Inheritance, access, extends	Reuse as well as replace existing rules/ Tight coupling and possible overriding conflicts
RubyTL	Phasing, refinement rules	Easy to obtain strict refinement/ Overriding behaviour of phasing can be difficult to understand
CT	Logic sequencing	In-place transformation allows free rule interaction/ Rule interaction easily becomes complex and can generate conflicts
VIATRA2	Module import, pattern invocation	Pattern invocation allows for fine-grained reuse/ Unclear whether import has overriding semantics

Table 1. Comparison overview of composition mechanisms in transformation languages.

an abstract syntax based virtual machine, in which the instructions are organised in an abstract syntax graph (i.e. an EMF model). Instructions communicate with each other via named local variables.

IDC is a *continuation*-based virtual machine: lists of instructions are grouped in continuations – executable blocks that can be interrupted and resumed – that are entered into an execution queue. Instructions themselves appear to be communicating via a stack. The continuation-based approach eliminates the need for multiple phases in the model transformation algorithm, as each continuation waits for any required data before resuming.

The current ATL virtual machines – regular VM and EMFVM – are stack-based virtual machines, where instructions are organised in a flat list and communicate with each other via the stack. Local variables are supported by special instructions: LOAD and STORE. Whereas both ATC and IDC formats are represented as EMF models, ATL bytecode is stored as a proprietary XML file.

3 EMF Transformation Virtual Machine

The EMF Transformation Virtual Machine (EMFTVM) is derived from the current ATL VMs and bytecode format. However, instead of using a proprietary XML format, it stores its bytecode as EMF models, such that they may be manipulated by model transformations. A special EMF resource implementation allows EMFTVM models to be stored in binary format, which is faster to load and save, and results in smaller files.

Apart from the standard ATL bytecode primitives, such as modules, fields, and operations, EMFTVM bytecode includes rules and code blocks. Fig. 1 shows the structure of rules and code blocks. Code blocks are executable lists of instructions, and have a number of local variables and a local stack space. Operation bodies and field initialisers are represented as code blocks in EMFTVM. Code blocks may also have nested code blocks, which can be manipulated and invoked from its containing block. These nested code blocks therefore effectively represent *closures*, which are nameless functions that can be passed as parameters to other functions. Closures are helpful for the implementation of OCL's higher-order operations, such as `select` and `collect`, which are parametrised by nested OCL expressions.

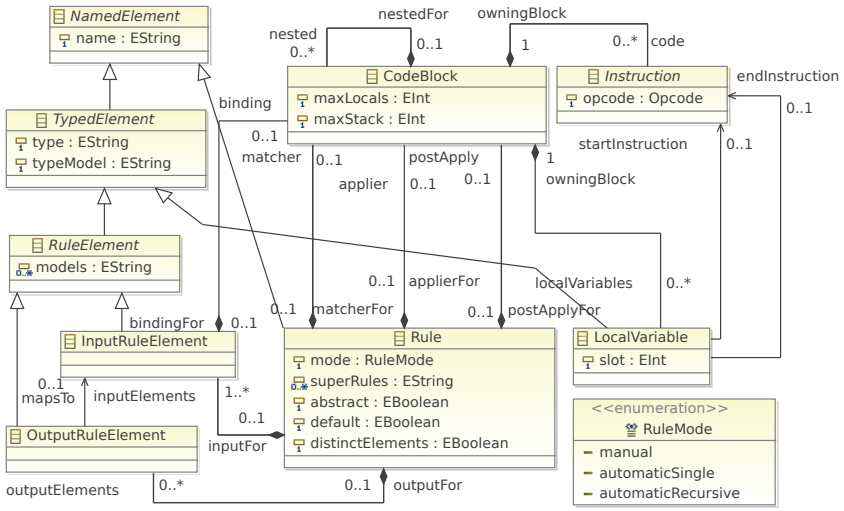

Fig. 1. Structure of EMFTVM rules and code blocks.

Rules consist of input and output rule elements, a matcher code block, applier code block, and post-apply code block. The matcher code block takes potential input element candidates as parameters, and returns a boolean value, representing a match. The applier code block takes the input and (newly created) output

elements as parameters, and assigns the bindings of the output elements. The post-apply code block also takes the input and output elements as parameters, and performs any (imperative) post-processing specified in the rule. Execution of rules is therefore done in three phases: (1) **matching**; only input elements are guaranteed to be present, (2) **applying**; all output elements and traces are guaranteed to exist, but no bindings may have been applied, (3) **post-apply**; all input and output elements, traces, and bindings are guaranteed to be present.

Rules can be invoked manually, automatically, and recursively automatically. Manual rules correspond to ATL lazy rules (and called rules). Automatic rules correspond to ATL matched rules. Recursively automatic rules do not apply to ATL, and may be ignored for the purpose of this paper. Rules can also be marked as default, which causes that rule to create default traces. Default traces can be looked up using ATL's implicit tracing mechanism, and only one default trace may exist for any given source pattern. Non-default traces are just stored in the trace model, and are not used by the EMFTVM transformation engine.

Rules can have a number of super-rules, which are stored by name. This decision allows EMFTVM to resolve and link the super-rules of each rule at load-time, whereas storing a super-rule reference would have hardcoded the super-rule in the bytecode. This is comparable to how the Java VM does super-class lookup. Finally, rules can be marked as abstract, which means that they are only applied as part of a non-abstract sub-rule, but never by themselves.

To summarise: by explicitly representing rules in the bytecode, rule inheritance can be resolved at load-time. As a consequence, rules stored in imported modules can be taken into account, and super-rules can be redefined by module superimposition before the reference to the super-rule is resolved in the sub-rules. This solves the historic mismatch between ATL's rule inheritance and module superimposition.The following sections will discuss in detail how rule inheritance and module superimposition are implemented in the EMFTVM, while preserving the original goals of rule inheritance and module superimposition: optimised matching of rule inheritance hierarchies and incremental compilation of modules.

4 Rule inheritance

ATL's rule inheritance has been extended to support multiple inheritance, and adding extra input elements in sub-rules. 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 element variables is required.

Fig. 2 outlines the semantics for rule matching in the context of rule inheritance. Each rule is represented by a box with compartments. The upper left compartment contains the input elements, whereas the upper right compartment contains the output elements. Furthermore, three more compartments exist to indicate that each rule has a matcher, applier, and post-apply code block associated with it.

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 name: 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.

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. This guarantees the safety of the load-time rule inheritance mechanism: the super-rule may be replaced with *any* other super-rule, while the integrity of the sub-rule remains intact. If a super- and sub-rule match on completely different elements, they will simply not produce any combined match, and the sub-rule is never applied.

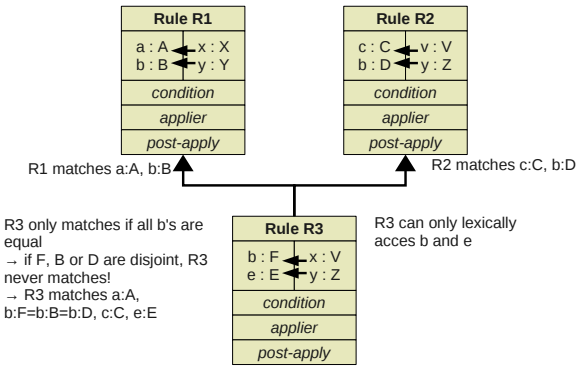

Fig. 2. Matching semantics for rule inheritance.

Fig. 3 outlines the semantics for rule application in the context of rule inheritance. The arrows between output elements and input elements in the figure represent **mapsTo** annotations. For the purpose of ATL's implicit tracing mechanism, such annotated output elements will be returned by the tracing resolver, whenever the corresponding input element is being resolved. Such **mapsTo** annotations can be overridden in sub-rules. In R3, x maps to b, which overrides the x maps to a annotation in R1. Similarly, y maps to b in R2 is overridden by y maps to e in R3. The annotation v maps to c in R2 is inherited as is by R3.

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 is 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 type specified in the most specific rule that matched, i.e. the sub-rule. Therefore, that type must be *co-variant* with the type specified for the same element 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.

Finally, 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, i.e. manual, automatic, or recursively automatic, and creating the same kind of trace, i.e. default or non-default.

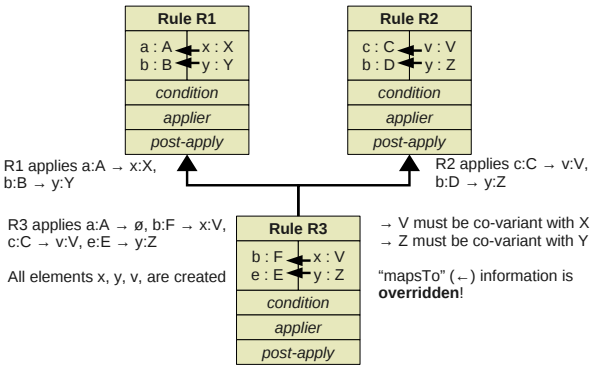

Fig. 3. Application semantics for rule inheritance.

The (automatic) rule matching algorithm performs optimised matching[2] 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 works as follows:

1. All rules without super-rules are matched, and their matches (tuple of input elements) are stored.
2. All rules for which the super-rule has matched any elements are now matched, and their matches are stored. For all matches, the super-rule match is removed.
3. The previous step is repeated until no applicable rules exist, or they do not match against any elements.

[2] As opposed to naive, full model iteration for all (automatic) rules.

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 and **mapsTo** information.
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.

This algorithm ensures that sub-rules are only matched for the elements that have already been matched by their super-rules, and does no unnecessary matching against remaining model elements. It also ensures that sub-rules cannot widen the initial input element type constraints and constraints encoded in the matcher code block of the super-rules.

5 Module superimposition

If we apply the semantics of ATL's current module superimposition mechanism, this will already interact well with the rule inheritance semantics we have defined in section 4: non-existent rules, fields and operations are added, while existing rules, fields and operations are redefined. However, module superimposition stacks must be defined for each launch configuration, and any **uses** clause in the original ATL file is no longer present in the ATL bytecode.

EMFTVM changes this: each EMFTVM module has a number of imported modules. This enforces that (1) all dependencies are loaded, and that (2) they are superimposed in the right order. Fig. 4 shows how this 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. Note that circular imports – and self-imports – are considered incorrect.

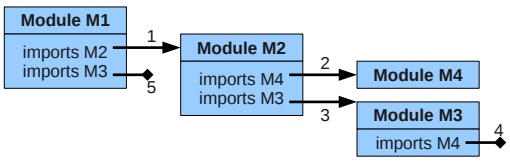

Fig. 4. Module import semantics.

Whenever a module is imported, its field, 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. Therefore, overloading of fields and operations with different context/parameter types is supported in EMFTVM. Fields and operations are only redefined by a superimposed module if the context and parameter types match.

Because rules are only registered by name, any rule with the same name may redefine an existing rule when superimposed. That means additional constraint checking is required for rule redefinition. Rules must be of the same kind to allow sound redefinition. Also, any type safety checks done for rule inheritance must be done again after redefinition.

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`, `M1` imports `M4`, and can redefine elements of `M4`.

6 Tool support

EMFTVM is implemented as an Eclipse feature[3]. It provides an implementation of the EMFTVM bytecode metamodel, including its interpreter, and a compiler for ATL. The compiler can be activated for specific ATL modules by specifying "`-- @atlcompiler emftvm`" on the first line. An ATL builder extension picks up this annotation, and invokes the EMFTVM compiler on the corresponding ATL file.

The compiler for ATL is implemented in ATL, and consists of two stages: an `ATLtoEMFTVM.atl` mapping transformation[4] and an `InlineCodeblocks.atl` transformation[5] that in-lines unnecessary nested code blocks into its container code block. This way, the mapping transformation can follow the exact same nesting structure of the source `.atl` model in the target `.emftvm` model. That makes `ATLtoEMFTVM.atl` a simple, one-to-one mapping transformation. The `InlineCodeblocks.atl` transformation then improves performance by reducing any unnecessary code block nesting to monolithic, in-lined code blocks.

Some exploratory observations, where we execute the ATL-to-EMFTVM compiler both in ATL's EMF VM as well as in the new EMFTVM, have shown that performance of the current EMFTVM implementation and its compiled code is about half that of the current EMF VM for ATL. It is therefore expected that EMFTVM is faster than ATL's regular VM, as ATL's EMF VM is at least 5 times faster than the regular VM.

[3] http://soft.vub.ac.be/soft/research/mdd/emftvm

[4] http://soft.vub.ac.be/viewvc/EMFTVM/trunk/emftvm.compiler/transformations/ATLtoEMFTVM.atl?view=markup

[5] http://soft.vub.ac.be/viewvc/EMFTVM/trunk/emftvm.compiler/transformations/InlineCodeblocks.atl?view=markup

7 Conclusion and future work

This paper has presented a revised semantics for ATL's rule inheritance and module superimposition composition mechanisms. These revised semantics tackle the historic problem of not being able to combine rule inheritance and module superimposition in ATL. This is achieved by defining both at the same level: in the virtual machine.

For this purpose, a new virtual machine and corresponding bytecode format have been developed, called the EMF Transformation Virtual Machine (EMFTVM). The EMFTVM explicitly represents transformation rules in its bytecode format, including the inheritance hierarchy information. Therefore, the EMFTVM can apply rule inheritance at load-time, at the same time that module superimposition takes place. Rule inheritance can therefore make use of rules that have been redefined as a consequence of module superimposition.

The implementation of the rule matching algorithm in the EMFTVM respects the original goal of rule inheritance, which is to enable optimised matching. It does this by first matching super-rules, and only matching sub-rules on elements that were previously matched by its super-rules. This algorithm is an extended version of the original algorithm, and supports multiple inheritance.

Several soundness and safety constraints have been given for rule inheritance as well as module superimposition. These constraints are not yet checked by the EMFTVM implementation. This will be added in the near future.

A constraint check that has not been discussed in this paper is the conflicting output element binding constraint. This conflict occurs where multiple super-rules want to bind the same output element property. In its current implementation, the EMFTVM will apply the bindings in the order that the rule inheritance has been specified. In the future, the EMFTVM may be extended to detect such conflict situations.

Preliminary performance observations have shown that EMFTVM does not match the performance of the current ATL EMF VM. In time, performance of EMFTVM may be improved through more aggressive inlining of code blocks, and further optimisation of the matching algorithm. Furthermore, a detailed performance analysis, including scaling properties, may be done in the future.

References

1. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT 2008), Zürich, Switzerland. Lecture Notes in Computer Science, vol. 5063, pp. 46–60. Springer-Verlag (Jul 2008)
2. Kurtev, I., van den Berg, K., Jouault, F.: Rule-based modularization in model transformation languages illustrated with atl. Science of Computer Programming 68(3), 111–127 (2007)
3. Sánchez-Barbudo, A., Sánchez, E.V., Roldán, V., Estévez, A., Roda, J.L.: Providing an open virtual-machine-based QVT implementation. Actas de los Talleres de las Jornadas de Ingeniera del Software y Bases de Datos 2(3), 42–51 (2008)

4. Schürr, A.: Specification of graph translators with triple graph grammars. In: Tinhofer, G. (ed.) Proceedings of the WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science, vol. 903, pp. 151–163. Springer-Verlag (1994)
5. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software and Systems Modeling 9(3), 285–309 (Oct 2009)
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: Accepted for the 4th International Conference on Model Transformation (ICMT 2011) (Jun 2011)
7. Yie, A., Wagelaar, D.: Advanced traceability for ATL. In: Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL 2009), Nantes, France. pp. 78–87 (Jul 2009)