Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Dennis Wagelaar · Ragnhild Van Der Straeten · Dirk Deridder

Module superimposition: a composition technique for rule-based model transformation languages

Received: date / Accepted: date

Abstract As the application of model transformation becomes increasingly commonplace, the focus is shifting from model transformation languages to the model transformations themselves. The properties of model transformations, such as scalability, maintainability and reusability, have become important. Composition of model transformations allows for the creation of smaller, maintainable and reusable transformation definitions that together perform a larger transformation. This paper focuses on composition for two rule-based model transformation languages: the ATLAS Transformation Language (ATL) and the QVT Relations language. We propose a composition technique called module superimposition that allows for extending and overriding rules in transformation modules. We provide executable semantics as well as a concise and scalable implementation of module superimposition based on ATL.

Keywords Software Engineering · Model Driven Engineering · Model Transformation

This work is part of the VariBru project, which is funded by the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB).

This work is also funded by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy.

Dennis Wagelaar
Vrije Universiteit Brussel
Tel.: +32-2-6292974
Fax: +32-2-6292870
E-mail: dennis.wagelaar@vub.ac.be

Ragnhild Van Der Straeten
Vrije Universiteit Brussel
E-mail: rvdstrae@vub.ac.be

Dirk Deridder
Vrije Universiteit Brussel
E-mail: dirk.deridder@vub.ac.be

1 Introduction

The application of model transformation has become increasingly commonplace in model-driven engineering, with a number of stable model transformation languages and tools available. The survey performed by Czarnecki et al. in [5] already covers 30 different model transformation approaches. The Object Management Group has even released the MOF Query/View/Transformation (QVT) standard transformation language [16]. Whereas the main focus of the model transformation community initially lay on the expressiveness of transformation languages, other properties are starting to become important, such as scalability, maintainability and reusability of model transformation *definitions*. The term model transformation *definition* refers to an expression in a model transformation language and is intended to disambiguate from the term model transformation *execution*, which refers to the act of transforming models [8]. As model-driven engineering becomes more mature, the model transformation definitions used typically become more elaborate. During the evolution of a model transformation definition, alternatives to the standard transformation scenario are discovered. An example of such an alternative is that we want to generate different getter and setter operations for ordered and non-ordered properties with a multiplicity higher than one. Such alternatives are integrated back into the original model transformation definition, which will grow in size and complexity. This increase in size and complexity is detrimental to maintainability and reusability of model transformation definitions, especially where multiple complex transformation definitions are involved. Such complex transformation definitions may include general rules that occur in other transformation definitions as well, while at the same time they may include very specific rules that apply only to that specific transformation definition.

In order to keep such model transformation definitions maintainable, they eventually have to be split up into separate model transformation definitions of a manageable size and scope. By this, we mean that the rules in each transformation definition are highly reusable and have a high degree of cohesion. Those separate model transformation definitions have to be composable in order to achieve the intended transformation result. By composable model transformation definitions, we mean that their specified behaviour can be composed into a well-defined, combined behaviour. The concept of composable modules is already well-known in the domain of programming, where programs are also modularised in order to cope with their (essential) complexity. In [4], Cuadrado et al. also argument how factorisation and subsequent composition of transformation definitions benefits reusability. This paper includes a number of specific model transformation scenarios that aim to demonstrate how model transformation definitions can benefit from modularisation.

Perhaps the most straightforward method of composition is to chain several model transformation executions together by providing the output of one transformation execution as input for another transformation execution. Another method is to compose the rules from a number of transformation definitions, resulting in a single, combined transformation execution. The latter method typically requires the model transformation definitions that will be composed to be expressed in the same language. This is because the transformation language semantics must be aligned in order to combine the parts of several model transformation definitions. In a workshop on the topic of model transformation composition [9], these two methods were labelled as *external* and *internal* transformation composition, respectively. We believe that both composition methods are necessary and complement each other, as we will also illustrate in this paper under subsection 4. Internal and external composition do not interfere with each other: internal composition combines model transformation definitions into a model transformation execution, whereas external composition combines model transformation executions.

The focus of this paper is on a technique for internal transformation composition, which means that the composition technique is specific to the domain of a particular transformation language. We propose a composition technique called *module superimposition*. Module superimposition allows one to overlay several transformation definitions on top of each other and then execute them as one transformation. We will discuss our composition technique based on the ATLAS Transformation Language (ATL) [7], which is used as an implementation vehicle for our experiment. We provide an executable semantics for ATL module superimposition in the form of a higher-order transformation expressed in ATL. In addition, we provide a concise and scalable implementation of ATL module superimposition, based on the ATL virtual machine. As a secondary target, we discuss module superimposition in the context of the QVT Relations [16] language. By translating our composition technique to QVT Relations, we assess whether our composition technique can be incorporated in model transformation languages other than ATL.

This paper is an extension of our ICMT 2008 conference paper [18]. In this paper, we provide a more in-depth discussion of ATL module superimposition semantics, we explain in detail how ATL module superimposition is implemented in a concise and scalable way, we reflect on limitations in the QVT Relations specification and tools that affect the application of module superimposition to QVT Relations, we provide a detailed comparison to related work and we provide an updated outlook on future work.

The rest of this paper is organised as follows: in section 2, we briefly explain ATL. After that, in section 3, we introduce module superimposition based on ATL and we also discuss how module superimposition interacts with other composition techniques in ATL. In section 4, we provide a number of usage scenarios for module superimposition. We then discuss module superimposition semantics in section 5 by means of a higher-order transformation that performs module superimposition. We then explain in section 6 how module superimposition is implemented in ATL and which optimisations have been made. We also discuss briefly in section 7 how module superimposition applies to QVT Relations. After that, section 8 discusses related work, followed by future work in section 9 and the conclusion in section 10.

2 ATLAS Transformation Language

ATL is a MOF-based transformation language that combines declarative rules with imperative statements. ATL makes a strict distinction between input models and output models: a model cannot serve as both input model and output model in a single transformation execution. Input models are read-only, while output models are write-only. Output models are always empty at the start of a transformation execution and cannot be navigated during the transformation execution. One can refer to elements in an output model via the *implicit tracing mechanism*, which is explained later in this section.

ATL supports three kinds of units: *libraries*, *queries* and *modules*. Libraries contain helper methods, which can be used in other ATL units. Queries define a read-only navigation over one or more input models and return a simple value. Queries can also contain helper attributes and methods. Modules consist of rules that transform input model elements into output model elements, and can also contain helper attributes and methods. ATL modules are the only kind of unit that can return output models. As our composition technique applies to ATL modules, we limit ourselves to discussing modules in the remainder of this section.

2.1 Modules

An ATL transformation module has a number of input models and typically one output model. It contains a number of *rules* that define the mapping from source elements to target elements. ATL has two kinds of rules: *matched* rules and *called* rules. Matched rules are automatically triggered, while called rules must be invoked from a matched rule. There also exists a special kind of matched rule that does not automatically trigger: the *lazy* rule. Hence, a lazy rule must be explicitly invoked, just like a called rule. The difference between lazy rules and called rules is that lazy rules have a matching specification, just like matched rules, whereas called rules have a parameter specification. We will limit ourselves to an explanation of matched rules, as this is sufficient to understand the principle of module superimposition. A complete discussion of ATL can be found in [6].

Listing 1 shows an example ATL module, named "UML2Copy", that copies a UML Model element to another UML Model element. The UML2Copy module has one output model named "OUT" of model type "UML2" and one input model "IN", which is also of model type "UML2". In ATL, models and model types are bound to concrete models and meta-models in a run configuration, which is not part of the module. Multiple run configurations can be defined for each ATL module. ATL does not perform any type-checking at compile-time and allows the developer to use any meta-class or property name. Only at run-time, ATL resolves meta-classes and properties by their name in the bound meta-model. In our example, the model type "UML2" is (intended to be) bound to the Eclipse UML2 meta-model. The relevant part of this meta-model is shown in Fig. 1.

```
module UML2Copy;
create OUT: UML2 from IN: UML2;
rule Model {
  from s: UML2!"uml::Model"
  to t: UML2!"uml::Model" (
    name <- s.name,
    visibility <- s.visibility,
    viewpoint <- s.viewpoint)
}
```

Listing 1 UML2Copy transformation module

The transformation module has one *matched* rule named "Model". ATL matched rules have a **from** part and a **to** part. The **from** part specifies which model elements from the input model(s) trigger the matched rule. The **to** part creates one or more model elements in the output model. In the example, any instance of the meta-class "uml::Model" from the "UML2" meta-model triggers the rule, where the "uml::" prefix specifies that the "Model" meta-class is inside the "uml" package. ATL uses '<-' to specify assignment: the "name", "visibility" and "viewpoint" values of the source Model "s" are assigned to same properties of the target Model "t".

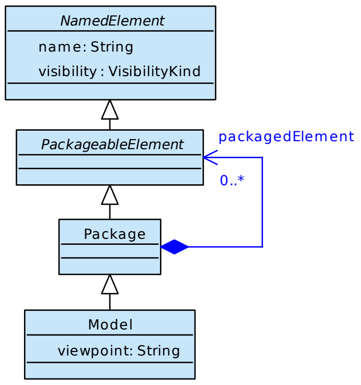


Fig. 1 Part of the Eclipse UML2 meta-model.

Listing 2 shows how multiple matched rules interact. The "Model" rule now includes an assignment of the "packagedElement" property. "s.packagedElement" refers to a collection of packaged model elements in the source model. Each of those model elements may separately match against a rule in the transformation module. Normally, "t.packagedElement" of the "Model" rule is not supposed to contain the elements from "s.packagedElement", because those elements reside in the source model. Rather, "t.packagedElement" should contain the "equivalents" of those elements in the target model. These "equivalents" are the elements created by a rule that matched against an element from "s.packagedElement". ATL's *implicit tracing mechanism* takes care of this and automatically translates assignments of source elements to their target element counterparts whenever those source elements trigger a matched rule in the transformation module.

```
module UML2ExtendedCopy;
create OUT: UML2 from IN: UML2;
rule Model {
  from s: UML2!"uml::Model"
  to t: UML2!"uml::Model" (
    name <- s.name,
    visibility <- s.visibility,
    viewpoint <- s.viewpoint,
    packagedElement <- s.packagedElement)
}
rule Package {
  from s: UML2!"uml::Package" (
    s.oclIsTypeOf(UML2!"uml::Package"))
  to t: UML2!"uml::Package" (
    name <- s.name,
    visibility <- s.visibility,
    packagedElement <- s.packagedElement)
}
```

Listing 2 UML2ExtendedCopy transformation module

This kind of source-to-target element tracing [5] is defined by the **from** element and the first **to** element. Tracing information for the other **to** elements is also recorded, but must be retrieved explicitly in ATL via the "`resolveTemp()`" API method. The exact workings of "`resolveTemp()`" are not relevant for the remainder of this paper. The tracing information is used to translate an assignment of source elements to target elements: "t.packagedElement" in the "Model" rule will not contain the elements of "s.packagedElement", but rather the (first) target elements that trace back to the elements of "s.packagedElement".

The "Package" rule copies all instances of "uml::Package" that satisfy the additional condition "`s.oclIsTypeOf(UML2!uml::Package)`". This additional condition is necessary to prevent the rule from triggering against subclasses of "uml::Package", such as "uml::Model".

Listing 3 provides another example transformation module to demonstrate ATL's implicit tracing mechanism[1]. The Class2Relational module translates classes and attributes to relational database tables and columns. The corresponding "Class" and "Relational" meta-models are shown in Fig. 2 and Fig. 3. All classes are translated to tables and all attributes with single values are translated to columns. All multi-valued attributes are translated to a separate table with an "id" column and a column that contains the attribute values.

```
module Class2Relational;
create OUT: Relational from IN: Class;
rule Class2Table {
  from c: Class!Class
  to t: Relational!Table (
    name <- c.name,
    col <- c.attr->reject(e|e.multiValued))
}
rule SingleValuedAttribute2Column {
  from a: Class!Attribute (
    not a.multiValued)
  to c: Relational!Column (
    name <- a.name)
}
rule MultiValuedAttribute2Column {
  from a: Class!Attribute (a.multiValued)
  to t: Relational!Table (
    name <- a.owner.name + '_' + a.name,
    col <- Sequence{id, value}),
  id: Relational!Column (
    name <- 'Id'),
  value: Relational!Column (
    name <- a.name)
}
```

Listing 3 Class2Relational transformation module

The interesting part of this transformation module is the "`col <- c.attr->reject(e|e.multiValued)`" assignment in the "Class2Table" rule. This assignment specifies that the columns of each class' table are made up out of the class' attributes that are not multi-valued. However, all attributes that are not multi-valued are transformed into columns by the "SingleValuedAttribute2Column" rule. Therefore, the implicit tracing mechanism translates the attributes specified by the "`c.attr->reject(e|e.multiValued)`" expression into the corresponding columns, resulting from the "SingleValuedAttribute2Column rule".

The order of execution for matched rules is not specified in ATL. Any execution order of the matched rules should result in the same transformation output. This is enforced in ATL by requiring that all matched rules in a module are *confluent* [14], which in the context of ATL means that no two rules are allowed to trigger on the same input. As the transformation input cannot be changed and the transformation output cannot be used as input (output cannot be read), the **to** parts need not be considered and non-overlapping **from** parts are sufficient to ensure rule confluence in ATL.

3 Module superimposition

ATL transformation modules are normally run one transformation module at a time. While ATL allows for factoring out helper methods in libraries, the various kinds of rules and helper attributes must always reside in an ATL module. This limits the reuse of transformation rules

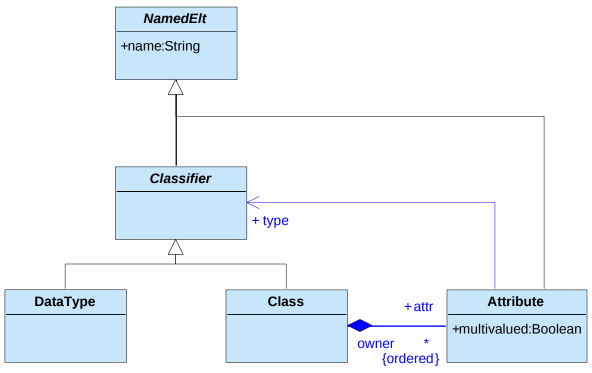


Fig. 2 The "Class" meta-model.

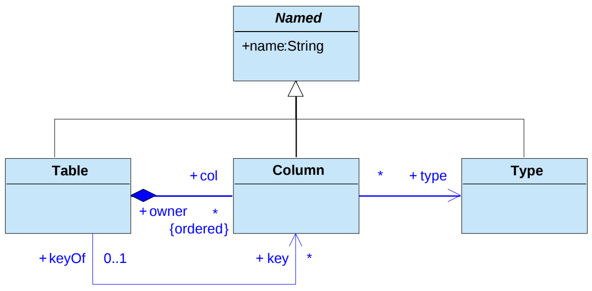


Fig. 3 The "Relational" meta-model.

[1] Source: http://www.eclipse.org/gmt/omcw/

to external composition, while helper attributes are not reusable at all. This limitation in reuse can lead to code duplication in transformation modules, which in turn increases the maintenance effort.

We propose to split up transformation modules into modules of manageable size and scope, which are then *superimposed* on top of each other. This results in (the equivalent of) a transformation module that contains the union of all transformation rules. In addition to adding rules, it is also possible for a transformation module to *override* rules from the transformation modules it is superimposed upon. By *overriding*, we mean *replacing* the original rule with a new one, whereby it is not possible to refer to the original rule anymore. This allows for rule-level adaptation of one transformation module by another and improves reusability of transformation modules. Other rule-based model transformation languages also support rule-level adaptation, such as QVT Relations and Operational Mappings [16], ETL [11] and RubyTL [4]. The QVT Relations language is covered in section 7, while the other languages are discussed in section 8.

Rule overriding is done by *name*: superimposed rules with the same name as an existing rule override the existing rule. This means that there is no *obliviousness* between the modules in a superimposition, where modules do not need to consider the contents of other modules. The superimposing module typically needs to know about the rule names and content in the base module, such that it can override rules in a purposeful manner. ATL's implicit tracing mechanism makes that rules can depend on the result of other rules in the transformation module (see subsection 2.1). In order to safely override a rule, the overriding rule should (1) match a superset of the elements that were matched by the overridden rule, and (2) create a superset of the output elements, which have to be of the same or more specific (substitutable) types, compared to the overridden rule. Explained in terms of covariance and contravariance: (1) the matching specification of the overriding rule must be *covariant* with respect to the original matching specification, while (2) the types of the output elements of the overriding rule must be *contravariant* with respect to the original output elements. In this way, substituting traces are available for all original traces after overriding, and their target elements can substitute the original target elements. Safe rule overriding is only required if there are rules that depend on the output of the overridden rule.

In a large transformation system, rule naming conventions and/or a special management system for rule naming may be required to perform meaningful superimpositions. Such a naming convention scheme could use the meta-class – or type – name in the rule matching specification as a basis for the rule name, while adding a descriptive term for any OCL restriction that is part of the rule's matching specification.

Rule overriding is *not* done on the basis of the rule's matching specification, as it quickly becomes unclear to the developer which rule overrides which other rule and for which subset of the input elements. Rule overriding based on matching specification would also allow the overriding of multiple rules by a single rule, which makes safe rule overriding even more difficult to accomplish. Rule overriding based on matching specification would also limit the ways in which one can modify the matching specification in the overriding rule. It is not possible, for example, to reduce the set of elements on which a rule matches by overriding that rule. While that is not safe rule overriding, it is a useful way to simulate deletion of model elements in ATL.

In the case where partial overriding behaviour on the basis of matching specifications is required, ATL's rule inheritance mechanism provides a better alternative. Rule inheritance requires that the developer makes his/her intention to override another rule explicit by adding an inheritance clause. In addition, the matching specification of the sub-rule indicates for which subset of input elements the sub-rule should be applied in favour of the super-rule. The relationship between module superimposition and rule inheritance is discussed in depth in subsection 3.1.

Fig. 4 shows an example of a typical use case for superimposition: the transformation rules of a general module are reused and overridden where necessary by specific modules that specify a slight variation in the base behaviour. In this case, the transformation rules of the UML2Copy transformation module are reused and overridden where necessary by the UML2Profiles transformation module. While the UML2Copy transformation module given earlier in this paper contains only one rule, the real UML2Copy is generated from the UML metamodel and includes a transformation rule for every meta-class of which it must copy the instances[2]. This amounts to approximately 200 rules for the entire UML2 metamodel. Any refinement transformation basically needs to copy all meta-class instances, except for the few meta-class instances that are refined. The UML2Profiles transformation module applies a profile to the "uml::Model" instance, provided it was not yet applied. All other elements should just be copied. To achieve this, the UML2-Profiles module is superimposed on the UML2Copy module. It overrides the existing "Model" rule, which copies each "uml::Model" instance, by a version that checks whether the profile we want to apply has already been applied. It also introduces a new rule "ModelProfile", which checks that the profile we want to apply has **not** been applied and then applies the profile. The resulting transformation module contains all rules from Fig. 4 that are not struck out.

As explained in section 2, ATL has a number of other constructs besides *matched* rules, such as *lazy* rules, *called* rules, helper *attributes* and helper *methods*. Simi-

[2] http://tinyurl.com/uml2copy

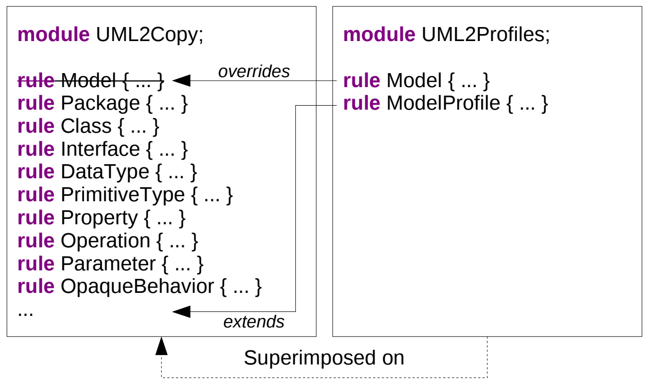


Fig. 4 ATL superimposition example.

lar to matched rules, all of these constructs have a *name* that is registered in a global ATL namespace during execution. Module superimposition therefore also applies to all these constructs. Note that attribute and method helpers also have a *context* in addition to their name: multiple helpers with the same name can exist as long as they have a different context. This is taken into account by module superimposition, which overrides helpers by name *and* context.

It is possible to superimpose more than one module. The result of such a superimposition depends on the order in which the modules are superimposed, in the same way that the order of functions matter in sequential function composition, for example $f(g(x))$. In the case of multiple superimposed modules, the first module is superimposed on top of the base module, after which the next module is superimposed on top of the result of the previous superimposition, and so on. If more than one superimposed module overrides the same rule, the rule in the module that was superimposed last will be in effect. The exact semantics of multiple module superimposition are discussed in section 5.

3.1 Interaction with other composition techniques

Module superimposition interacts with other composition techniques in ATL, such as helpers and called rules. In addition to the normal matched rules in ATL, module superimposition also allows for reusing and overriding called rules and helpers.

Called rules allow for functional composition in ATL. Called rules can be invoked (with side-effects) and return a value. With module superimposition, it is possible to replace parts of the function invocation chain by overriding called rules. It is also possible to invoke called rules from other modules in the superimposition stack. This introduces dependencies on the other modules, however, and should be used with care. It is advisable to limit invocation of called rules in other modules to the modules "below" (i.e. modules that are superimposed upon, not the superimposing modules). Any direct reference to called rules (or helpers) in other modules already requires those modules to be specified in the **uses** clause. This way, dependencies between modules are made explicit. The **uses** clause is normally reserved for libraries in ATL, since ATL could not combine multiple modules in one transformation execution before the addition of module superimposition. In the presence of module superimposition, the **uses** clause becomes meaningful for modules as well. The **uses** clause indicates what direct (lexical) dependencies the current ATL unit (module/library/query) has to which other units. Similar to model and model type specifications in ATL modules, the **uses** clause refers to names of referenced units rather than concrete units. The concrete referenced units are specified as part of a run configuration, and it is therefore possible to use different library/module implementations to satisfy a given **uses** clause.

Module superimposition has a similar effect on helpers as on called rules. Helpers are different from called rules in that they can have a context, however. The ATL engine keeps track of helper attributes and methods per context. That way, it is possible to define multiple helpers with the same name and a different context. Depending on the context, the corresponding version of the helper is used. As a consequence, superimposition can override helpers per context in ATL, leaving helpers with another context in place.

ATL supports another composition construct called *rule inheritance* [12]. Rule inheritance allows one to define general transformation rules that can be extended by specific rules. A sub-rule is required to specify a **from** part that matches the same or less elements than its super-rule. It can then inherit the **to** part from its super-rule and add its own entries to the **to** part. In ATL's current implementation, it not possible to separately superimpose sub- and super-rules. Only the sub-rules can be manipulated by module superimposition. This is because the ATL compiler in-lines the super-rules into the sub-rules, and no super-rules exist in the ATL bytecode. As module superimposition is performed *after* the compiler does its work, the super-rules are no longer available. In principle, module superimposition can be combined with rule inheritance without this limitation. If super-rules and sub-rules exist in the ATL bytecode, module superimposition can add new sub-rules and override existing super- and sub-rules. Section 5 already describes the semantics for this combination.

4 Superimposition usage scenarios

Module superimposition can be used in a number of specific model transformation scenarios. In our experiments with ATL, we have identified three of these scenarios:

Variation of base behaviour – a base module specifies the bulk of the transformation behaviour, while various smaller, superimposed modules specify variations of the base behaviour.

Factorisation of common behaviour – common behaviour in two or more transformations is factored out into a separate module.

Factorisation of meta-model-specific behaviour – whenever a meta-model spans multiple packages, models or files, the transformation rules can be spanned over multiple transformation modules accordingly.

Similar usage scenarios are also identified and explained in detail by Cuadrado et al. in [4], who provide a solution based on the RubyTL transformation language. RubyTL will be discussed further in section 8. In the following subsections, we will give an ATL example for each of the scenarios.

4.1 Variation of base behaviour

Module superimposition can be used to achieve a specific "base behaviour" of the transformation engine, such as copying the input model to the output model. In addition to ATL's standard and "refining mode" base behaviour, which performs an implicit copy from input to output model, superimposition can deal with non-standard situations, such as having multiple input (and/or output) models and filter which elements must be copied. Smaller transformation modules can be superimposed on top of the base behaviour module to specify incremental variations in transformation behaviour. Our example UML2-Copy transformation is meant to only copy elements from the model "IN" to the model "OUT". Listing 4 shows the UML2Profiles transformation module that is superimposed on UML2Copy.

UML2Profiles adds an extra input model, "ACCESSORS". The "ACCESSORS" model refers to a UML profile that is applied to the "OUT" model. The elements of the "ACCESSORS" model should not be copied, but should instead be referenced from the "OUT" model. This is achieved by checking that only elements contained in the "`inElements`" helper attribute match the **from** part from each rule. The "`inElements`" helper is provided by the UML2Copy module and contains all elements from "IN".

By separating the general copying functionality (UML2Copy module) from the specific refinement functionality (UML2Profiles module), we have achieved better maintainability, since it's much easier to find a specific transformation rule within a small, specific transformation module. This particular example also uses external composition by chaining together several refinement transformation steps. Module superimposition is used in each of the refinement steps. In this case, maintainability is also improved by reduced code duplication in all available refinement transformation modules; all copying code is now centralised and available for reuse. Finally, reusability is improved by the ability to extend and adapt general transformation modules, such as UML2Copy.

```
module UML2Profiles;
create OUT: UML2
from IN: UML2, ACCESSORS: UML2;
helper def: accessorsProfile:
UML2!"uml::Profile" =
  UML2!"uml::Profile".allInstances()
  ->select(p|p.name='Accessors')->first();
rule Model {
  from s: UML2!"uml::Model" (
    if thisModule.inElements->includes(s)
    then
      s.profileApplication->select(a|
        a.appliedProfile=
          thisModule.accessorsProfile)
      ->notEmpty()
    else false endif)
  to t: UML2!"uml::Model" (
    name <- s.name,
    visibility <- s.visibility,
    viewpoint <- s.viewpoint,
    profileApplication <-
      s.profileApplication)
}
rule ModelProfile {
  from s: UML2!"uml::Model" (
    if thisModule.inElements->includes(s)
    then
      s.profileApplication->select(a|
        a.appliedProfile=
          thisModule.accessorsProfile)
      ->isEmpty()
    else false endif)
  to t: UML2!"uml::Model" (
    name <- s.name,
    visibility <- s.visibility,
    viewpoint <- s.viewpoint,
    profileApplication <-
      s.profileApplication),
   pa : UML2!"uml::ProfileApplication" (
    applyingPackage <- s,
    appliedProfile <-
      thisModule.accessorsProfile)
}
```

Listing 4 UML2Profiles transformation module

4.2 Factorisation of common behaviour

Another usage scenario is the factorisation of common transformation behaviour, where multiple transformation definitions share a significant part of their behaviour. Fig. 5 shows an example of this scenario, where platform ontologies are generated from Java API models expressed in UML [19]. In this scenario, a number of Java API models are correlated to each other in terms of compatibility. The result is a set of OWL ontologies that model the Java API elements and how they relate to each other[3]. The arrows in the figure indicate which models are passed between the different transformation

[3] http://ssel.vub.ac.be/ssel/research:mdd:platformkit:ontologies

executions. The Jar2UML transformation execution converts Java class libraries in jar format into UML models. The UML2ToAPIOntology.atl transformation execution converts these UML models into OWL ontologies. The source UML model of the left-hand transformation chain is compared with the main source UML model in terms of compatibility: each pair of API elements from the two models is compared and determined to be incompatible, compatible, or equivalent (two-way compatible). The output OWL ontology of the left-hand transformation chain is used to create references from the main output OWL ontology to the left-hand OWL ontology. Different levels of granularity can be achieved for the OWL ontologies by superimposing either UML2ToPackageAPIOntology.atl or UML2ToClassAPIOntology.atl on top of the base UML2ToAPIOntology.atl module. This will result in either a package-level or a class- and interface-level ontology of the API elements. The base module contains general helper attributes that are used to store intermediate values, as well as a called rule that can be invoked from one of the superimposed modules. In this way, 152 lines of ATL code have been factored out into the UML2ToAPIOntology.atl module, while UML2ToPackageAPIOntology.atl and UML2ToClassAPIOntology.atl still consist of 112 and 132 lines of code respectively.

4.3 Factorisation of meta-model-specific behaviour

The third scenario deals with the situation where a meta-model spans over multiple files/packages/models. In other words: the meta-model is modularised into multiple, reusable parts. It makes sense to also modularise transformation definitions based on this meta-model, such that those transformation definitions are reusable together with their specific meta-model part.

Fig. 6 shows an example of this scenario, provided by the configuration language of our instant messenger case study[4], where the configuration language definition consists of two parts. The configuration language meta-model is split up in a general "Transformations" package and a specific "InstantMessenger" package. The model transformation definitions that make up the configuration generator follow this modularisation: the ConfigToBuildFile.atl transformation module has also been split up in two parts: one for each meta-model package, where ConfigToBuildFile.atl for "InstantMessenger" can be superimposed on ConfigToBuildFile.atl for "Transformations". The (matched) rules of left-hand ConfigToBuildFile.atl in Fig. 6 only refer to meta-classes from the left-hand "Transformations" meta-model, while the right-hand side ConfigToBuildFile.atl adds the rules that refer to the meta-classes of the right-hand "InstantMessenger" meta-model. This allows for reuse of the general "Transformations" infrastructure in other configuration languages and generators.

[4] http://ssel.vub.ac.be/ssel/research:mdd:casestudies

5 Semantics of module superimposition

This section provides an in-depth discussion of the semantics of module superimposition. An important aspect of the module superimposition semantics is that any combination of superimposed modules can be rewritten as a single transformation module. We have expressed the rewriting of two combined modules as a single module in a higher-order ATL transformation module. This higher-order transformation module serves as a basis for explaining the superimposition semantics. It also explains the semantics of superimposing more than one module. The superimposition of the first module on top of the base module results in a single, combined module. The next superimposed module is then superimposed on top of that single combined module. This is similar to a sequential function composition $f(g(x))$, where f is applied to the result of $g(x)$. As such, the effect of superimposing multiple modules is defined by applying the higher-order transformation module multiple times, taking the output of the previous superimposition as the new base module.

By expressing the semantics of ATL module superimposition in ATL itself, the semantics are executable and do not need any alignment with the ATL semantics. The actual implementation of ATL module superimposition, discussed in section 6, does not use this higher-order transformation module, but provides a more scalable and concise alternative based on the ATL virtual machine. The ATL meta-model, on which the higher-order transformation module operates, is given in Appendix A. The start of the higher-order transformation module is shown in Listing 5.

```
module Superimpose;
create OUT: ATL from IN: ATL, SUPER: ATL;
helper def: inElements:
Set(ATL!ATL::LocatedElement) =
  ATL!"ATL::LocatedElement"
    .allInstancesFrom('IN')
    ->reject(o|o.isOverridden())->asSet()
  ->union(ATL!"ATL::LocatedElement"
    .allInstancesFrom('SUPER')
    ->reject(s|
      if s.oclIsKindOf(ATL!"ATL::Rule")
      or s.oclIsKindOf(ATL!"ATL::Helper")
      then s.isOverriding()
      else false endif));
```

Listing 5 Superimpose transformation module

The higher-order transformation module superimposes the "SUPER" transformation module on the "IN" transformation module and writes the result into the "OUT" transformation module. It copies all the elements

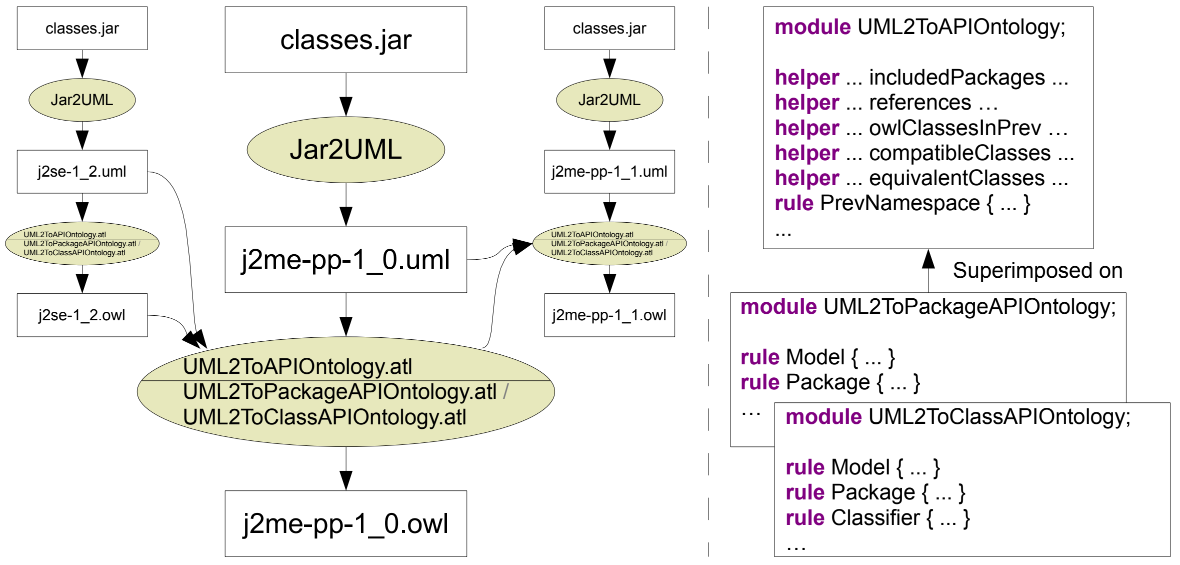

classes.jar
Jar2UML
j2se-1_2.uml
UML2ToAPIOntology.atl
UML2ToPackageAPIOntology.atl /
UML2ToClassAPIOntology.atl
j2se-1_2.owl
classes.jar
Jar2UML
j2me-pp-1_0.uml
UML2ToAPIOntology.atl
UML2ToPackageAPIOntology.atl /
UML2ToClassAPIOntology.atl
j2me-pp-1_0.owl
classes.jar
Jar2UML
j2me-pp-1_1.uml
UML2ToAPIOntology.atl
UML2ToPackageAPIOntology.atl /
UML2ToClassAPIOntology.atl
j2me-pp-1_1.owl
module UML2ToAPIOntology;
helper ... includedPackages ...
helper ... references ...
helper ... owlClassesInPrev ...
helper ... compatibleClasses ...
helper ... equivalentClasses ...
rule PrevNamespace { ... }
...
Superimposed on
module UML2ToPackageAPIOntology;
rule Model { ... }
rule Package { ... }
...
module UML2ToClassAPIOntology;
rule Model { ... }
rule Package { ... }
rule Classifier { ... }
...

Fig. 5 ATL superimposition scenario for generation of platform ontologies.

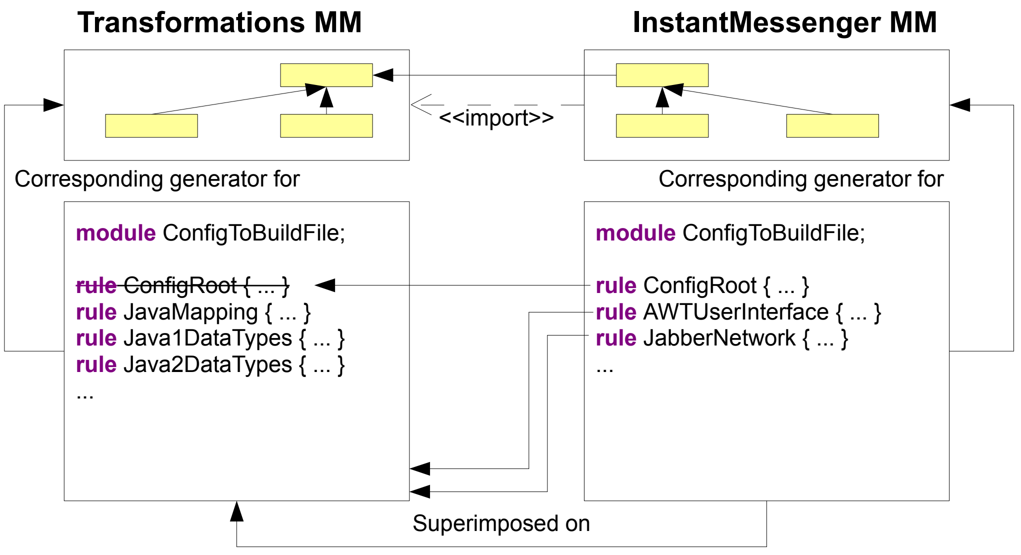

Transformations MM
InstantMessenger MM
<<import>>
Corresponding generator for
Corresponding generator for
module ConfigToBuildFile;
~~rule ConfigRoot { ... }~~
rule JavaMapping { ... }
rule Java1DataTypes { ... }
rule Java2DataTypes { ... }
...
module ConfigToBuildFile;
rule ConfigRoot { ... }
rule AWTUserInterface { ... }
rule JabberNetwork { ... }
...
Superimposed on

Fig. 6 ATL superimposition scenario for instant messenger configuration language.

in “`inElements`” directly to “OUT”. “`inElements`” is a helper attribute that contains all elements from “IN” that are not overridden, and all elements from “SUPER” excluding overriding rules and helpers. There are special transformation rules for overridden rules and helpers.

Listing 6 gives the definition of an overridden rule: each rule in the base module is overridden if and only if there exists a rule with the same name in the superimposed module and that rule is also of the same type. The rule type refers to the various kinds of rules available in ATL: matched rules, called rules and lazy rules (see also Appendix A). This means that the scope of overriding is always limited to the exact language construct: it is not possible to override matched rules by called rules, for example.

```
helper context ATL!"ATL::Rule"
def :isOverridden(): Boolean =
    ATL!"ATL::Rule"
    .allInstancesFrom('SUPER')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());
```

Listing 6 isOverridden helper for ATL::Rule

Rule deletion is not directly supported, but can be achieved by changing the condition of the **from** part. Overriding a rule by a rule that has a **from** part with the condition "`false`" has an equivalent effect to deleting that rule, since the rule can never trigger. This kind of rule "deletion" only applies to matched rules: lazy rules and called rules are triggered explicitly by an invocation. Enabling deletion of lazy/called rules is also dangerous, as it will invalidate any existing invocation of that rule. Deletion of matched rules is also not without risk, because of the effects it has on the implicit tracing mechanism. The affected source elements are no longer mapped to a target element, which may change the result of assignments in other rules.

The transformation rule in Listing 7 deals with overridden matched rules. Similar rules exist for overridden lazy rules, called rules and helper attributes/methods. The "`OverriddenMatchedRule`" transformation rule transforms the *overridden* matched rules from "IN" to "OUT" using all the property values from the *overriding* matched rule "`o`". As a consequence, the output matched rule "`t`" will have all the properties of "`o`", but it will still occur in the same place in "OUT" as "`s`" did in "IN". This is achieved by ATL's implicit tracing mechanism, which maps all "`s`" references to "`t`" references.

```
helper def: realInElements:
Set(ATL!"ATL::LocatedElement") =
    ATL!"ATL::LocatedElement"
    .allInstancesFrom('IN');
rule OverriddenMatchedRule {
  from s: ATL!"ATL::MatchedRule" (
    if thisModule.realInElements
        ->includes(s) then
      s.oclIsTypeOf(ATL!"ATL::MatchedRule")
      and s.isOverridden()
    else false endif)
  using { o: ATL!"ATL::MatchedRule" =
    s.overriddenBy(); }
  to t: ATL!"ATL::MatchedRule" (
    name <- o.name,
    ...)
}
```

Listing 7 OverriddenMatchedRule transformation rule

This becomes clearer when looking at Listing 8, which shows the transformation rule that deals with the transformation module element. The "`Module`" transformation rule copies only the transformation module element from "IN". The contained elements are retrieved from the "SUPER" transformation module in the **using** part. They are then appended to the (ordered) list of existing elements. In the case of overridden matched rules, the overridden rule is already contained in "`s.elements`". After the assignment, "`t.elements`" contains the same ordered list, except that its elements are all mapped to their "OUT" counterparts by the implicit tracing mechanism. All references to overridden rules and helpers are treated in this way.

```
rule Module {
  from s: ATL!"ATL::Module" (
    thisModule.realInElements->includes(s))
  using {
    superElements:
    Sequence(ATL!"ATL::ModuleElement") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.elements
        ->select(e|not e.isOverriding()))
      ->flatten();
    superInModels:
    Sequence(ATL!"OCL::OclModel") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.inModels)->flatten();
    superOutModels:
    Sequence(ATL!"OCL::OclModel") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.outModels)->flatten();
    superLibraryRefs:
    Sequence(ATL!"ATL::LibraryRef") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.libraries)->flatten();
  }
  to t: ATL!"ATL::Module" (
    name <- s.name,
    ...,
    libraries <- s.libraries
      ->union(superLibraryRefs),
    inModels <- s.inModels
      ->union(superInModels),
    outModels <- s.outModels
      ->union(superOutModels),
    elements <- s.elements
      ->union(superElements))
}
```

Listing 8 Module transformation rule

While ATL semantics specify that the execution order of matched rules does not matter, ATL's meta-model does keep a record of the rule order. This order is also followed by the ATL execution engine. By taking into account the rule ordering in our higher-order transformation, an exact specification is given of the *structure* of the resulting module composition. This *structure* in turn defines the *behaviour* in the current ATL execution engine.

The "`Module`" transformation rule also specifies how models and model types in the base and superimposed module are dealt with. The superimposition result contains the union of input models and the union of output models, all with their corresponding model types. Listing 9 shows how the "`OclModelElement`" rule makes sure that all references to "OCL::OclModel" instances are resolved to either the original instance or the overriding instance, if there is one. It uses the "`overriddenBy`" helper to do so. The "OCL::OclModel" instances refer to input/output (meta-)models that are defined in a transformation module.

```
helper context ATL!"OCL::OclModel"
def: overriddenBy(): ATL!"OCL::OclModel" =
  let selfInSuper:
  Sequence(ATL!"OCL::OclModel") =
    ATL!"OCL::OclModel"
    .allInstancesFrom('SUPER')
    ->select(r|
      r.name = self.name and
      r.oclType() = self.oclType()) in
  if selfInSuper->isEmpty() then self
  else selfInSuper->first() endif;
rule OclModelElement {
  from s : ATL!"OCL::OclModelElement" (
    thisModule.inElements->includes(s))
  to t : ATL!"OCL::OclModelElement" (
    location <- s.location,
    commentsBefore <- s.commentsBefore,
    commentsAfter <- s.commentsAfter,
    name <- s.name,
    type <- s.type,
    model <- s.model.overriddenBy())
}
```

Listing 9 OclModelElement transformation rule

The full higher-order transformation is split up in two modules: ATLCopy.atl and Superimpose.atl, where ATLCopy.atl is a simple copying transformation and Superimpose.atl provides the special transformation rules for superimposition. For the full source code, see Appendix B. As a proof of concept, Superimpose.atl is superimposed on ATLCopy.atl using the regular superimposition implementation (see section 6) and then applied to ATLCopy.atl and itself. The result is a single transformation module, ATLSuperimpose.atl, that represents the composition of ATLCopy.atl and Superimpose.atl. We have then verified that applying ATLSuperimpose.atl to ATLCopy.atl and Superimpose.atl yields the same results as the regular superimposition implementation applied to ATLCopy.atl and Superimpose.atl by comparing the results. This is not a guarantee that the implementation **always** yields the same results as the specification given here, but it serves as a test case for checking that the superimposition implementation follows the specification for a given input.

6 Implementation of module superimposition

While the semantics we have provided in section 5 are executable, they are not optimal as an implementation of module superimposition for ATL. One of the reasons for using module superimposition was to improve scalability; each time a change is made in a single transformation module, only that module needs to be recompiled. If this module is part of a superimposition composition, all modules involved must first be composed and the resulting module must then be compiled. In order to achieve improved scalability, module superimposition must be performed on the compiled ATL modules. That way, module superimposition has no effect on the compiler workload. This section explains how module superimposition is implemented in ATL and demonstrates how module superimposition can be implemented in a scalable and concise way in the ATL virtual machine.

Module superimposition is implemented as a load-time construct: there is no real transformation module that represents the result of superimposing several modules on top of each other, as was the situation in the previous section. Instead, several modules are *loaded* on top of each other, overriding existing rules and adding new rules. This approach leverages the natural semantics of extending and overriding in the ATL loading mechanism.

Fig. 7 shows a schematic overview of the superimposition implementation in the ATL virtual machine, and shows how the "ASM" representations of a base module and a superimposed module are loaded in sequence into ATL's execution environment. ATL transformation modules are first compiled to "ASM" instruction (or bytecode) format. ASM files contain operations with signatures, where a signature consists of a name, context type and a list of parameters. The body of each operation consists of a list of ASM instructions. Helper methods and helper attribute initialisation expressions are represented as operations in ASM format. Lazy and called rules are represented as operations with context type "Module". Matched rules are represented by two operations: one for matching and one for applying the rule, both with context type "Module" (`__matchModel` and `__execModel` for the "Model" matched rule in Fig. 7).

The ATL execution environment allows for loading (and reloading) operations into its lookup table. Rules and helpers are represented by such operations in their compiled form. Therefore, it is possible to load additional rules and helpers and reload overriding rules and helpers into the VM's lookup table. As a result, the optimised implementation of module superimposition consists of 377 lines of Java code[5], whereas the combined higher-order transformation specification explained in section 5 (ATLCopy.atl and Superimpose.atl) consists of 1035 lines of ATL code. Module superimposition has been implemented in ATL since the end of 2006 and was included in the ATL release since version 2.0[6].

The remainder of this section is organised as follows: subsection 6.1 provides a technical description of the ASM loading procedure as implemented in the current ATL engine with module superimposition support. Subsection 6.2 discusses the safety issues related to the bytecode adaptation that is performed as part of the ASM loading procedure. Finally, subsection 6.3 discusses the benefits and drawbacks of ATL's module superimposition implementation.

[5] http://tinyurl.com/AtlSuperimposeModule-java
[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=156095

6.1 The ASM loading procedure

```
1  getasm
2  push OclParametrizedType
3  push #native
4  new
5  dup
6  push Collection
7  call J.setName(S):V
8  dup
9  push OclSimpleType
10 push #native
11 new
12 dup
13 push OclAny
14 call J.setName(S):V
15 call J.setElementType(J):V
16 set col
17 getasm
18 push ecore::EObject
19 push UML2
20 findme
21 push IN
22 call J.allInstancesFrom(J):J
23 set inElements
24 getasm
25 push TransientLinkSet
26 push #native
27 new
28 set links
29 getasm
30 call A.__matcher__():V
31 getasm
32 call A.__exec__():V
```

Listing 10 UML2Copy `main` operation instructions

Whenever an ATL transformation is launched, the models and base transformation module (in ASM format) are loaded first. Then, each ATL library is loaded and its operations are registered in the operation lookup table of the execution environment. After that, the operations of the base transformation are registered in the lookup table. From here on, the explanation assumes the output of the `atl2006` compiler. The situation is slightly different for the older `atl2004` compiler, but the principle is the same. Our implementation covers the output of both compilers.

At this point, each superimposed module is loaded in sequence. The ASM file for each module contains a `main`, `__matcher__` and `__exec__` operation that perform helper attribute initialisation and the application of the matched rules. For each superimposed module, the bodies of the `main`, `__matcher__` and `__exec__` operations already registered in the execution environment are adapted, starting with the `main` operation. The helper attribute initialisation code from the superimposed module is inserted in the registered `main` operation. The address offsets for branching instructions in the registered `main` operation are updated to reflect the inserted instructions. The `main` operation from the superimposed module is then removed.

Consider the instructions of the UML2Copy.atl `main` operation in Listing 10. Lines 1–16 represent boilerplate instructions that occur in every module. Lines 17–23 represent initialisation code for the "inElements" helper attribute. Lines 24–32 are again boilerplate instructions. The exact semantics of each instruction can be found in the ATL virtual machine specification [1], but this information is not required to understand the loading procedure. Any helper attribute initialisation code from a superimposed module should be inserted just before instruction 24. The recurring boilerplate instruction patterns are used to find the specific helper attribute initialisation code.

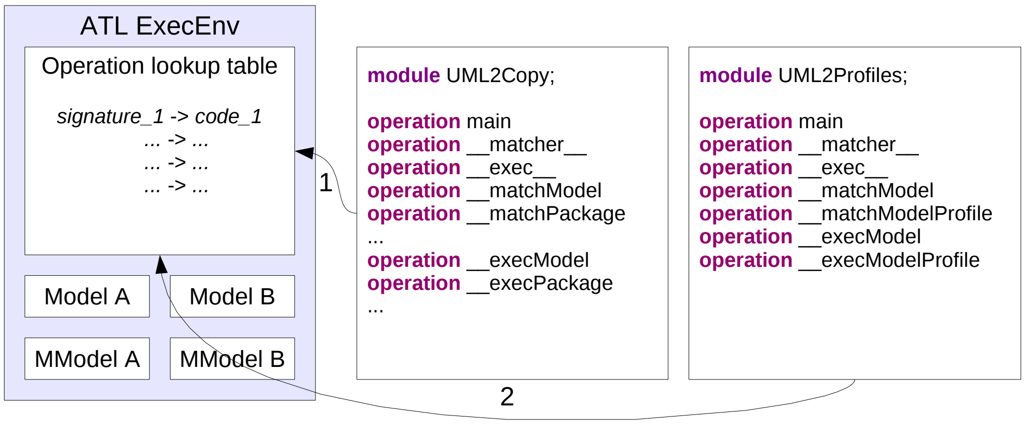
Now, the `__matcher__` and `__exec__` operations are adapted; these operations contain only instructions corresponding to invocations of the matching and application operations for each matched rule, such as "`__matchModel__`" and "`__execModel__`" in Fig. 7. Any such instructions corresponding to invocations that exist in the superimposed "`__matcher__`" and "`__exec__`" operations, but not in the registered operations, are *appended* to the registered operations. When the bodies of the registered "`__matcher__`" and "`__exec__`" operations have been adapted, the "`__matcher__`" and "`__exec__`" operations from the superimposed module are removed. Finally, the remaining operations of the superimposed module are registered. Note that the operation signatures for rules and helpers are different, even if the rule/helper names are the same. This way, name clashes between rules and helpers, for example, are avoided. Because of this, it is also impossible to override a rule by a helper (or vice versa) with module superimposition.

Listing 11 shows an extract of the UML2Copy.atl `__matcher__` operation instructions. This operation consists entirely of a repeated pattern of two instructions for each matched rule. These patterns are compared against the patterns in the `__matcher__` operation of the superimposed module. Any new patterns in the superimposed `__matcher__` operation are appended to the registered `__matcher__` operation.

```
getasm
call A.__matchEAnnotation():V
getasm
call A.__matchEStringToStringMapEntry():V
getasm
call A.__matchComment():V
...
```

Listing 11 UML2Copy `__matcher__` operation instructions

Listing 12 shows an extract of the UML2Copy.atl `__exec__` operation instructions. Similar to the `__matcher__` operation, this operation also consists entirely of a repeated pattern, this time consisting of ten instructions for each matched rule. These patterns are compared against the patterns in the `__exec__` operation of the superimposed module. Any new patterns in the superimposed `__exec__` operation are again appended to the registered `__exec__` operation.



module UML2Copy;

operation main
operation __matcher__
operation __exec__
operation __matchModel
operation __matchPackage
...
operation __execModel
operation __execPackage
...

module UML2Profiles;

operation main
operation __matcher__
operation __exec__
operation __matchModel
operation __matchModelProfile
operation __execModel
operation __execModelProfile

Fig. 7 ATL superimposition implementation.

```
getasm
get links
push EAnnotation
call NTransientLinkSet;.getLinksByRule(S):
    QNTransientLink;
iterate
store 1
getasm
load 1
call A.__applyEAnnotation(NTransientLink;):V
enditerate
getasm
get links
push EStringToStringMapEntry
call NTransientLinkSet;.getLinksByRule(S):
    QNTransientLink;
iterate
store 1
getasm
load 1
call A.__applyEStringToStringMapEntry(
    NTransientLink;):V
enditerate
...
```

Listing 12 UML2Copy __exec__ operation instructions

6.2 Safety of bytecode adaptation

The adaptation of instructions as we have just discussed comes with a risk. We make certain assumptions about the ATL compiler output: for example, each `main` operation starts with 16 boilerplate instructions, followed by a variable number of helper attribute initialisation instructions, followed again by at least 5 boilerplate instructions ending with `set links`. If we encounter other ASM code than expected, our superimposition implementation may produce an erroneous ASM instruction sequence.

One way to cover this risk is to require (and test for) a specific ATL compiler implementation. Any compiler change will immediately break our superimposition implementation. As the superimposition implementation only covers a small part of the generated ASM code, we have chosen to perform a number of local "sanity" tests on the generated ASM code. This allows for changes in the ATL compiler without breaking the superimposition implementation, as long as the local tests succeed. The superimposition implementation applies the following local sanity tests before it performs the superimposition:

1. Each `main` is at least 21 instructions long. The 21 is made up by a start boilerplate of 16 instructions that is contained in each transformation module, and an insertion point for instructions in the `main` that lies 5 instructions before the `set links` instruction.
2. Instruction 16 of each `main` is `set col`. This is a characteristic instruction that cannot occur anywhere else in the transformation module – and hence `main` – and the last instruction of the start boilerplate. This check is used as a partial verification that the expected start boilerplate is present.
3. There exists an instruction `set links` after instruction 16 for each `main`. This check is used to verify that we can properly recognise the end boilerplate by looking up the characteristic instruction (`set links` cannot occur anywhere else in a transformation module).
4. The first 16 instructions of the base `main` and the superimposed `main` are the same. This check augments check 2 and verifies that the start boilerplates of both `main` operations are the same.
5. Each `__matcher__` and `__exec__` contain only whole patterns of two respectively ten instructions long. This check verifies that `__matcher__` and `__exec__` consist of instruction runs of the same length that can be compared against each other and inserted as whole patterns (instead of dealing with each single instruction separately).
6. The starting instruction of each pattern in each `__matcher__` and `__exec__` is always the same. This check augments check 5 by checking for an indication that the instruction runs are in fact repetitive patterns.
7. The starting instructions of the patterns in a base and superimposed `__matcher__` or `__exec__` are always the same. This check further augments checks 5

and 6 by checking that both transformation modules use the same instruction patterns in `__matcher__` and `__exec__`.

6.3 Discussion

We mentioned earlier that the situation is slightly different for the `atl2004` compiler: it is not possible to use superimposition on a mix of `atl2006` and `atl2004` compiler output. The above sanity checks also detect this situation, as the first 16 instructions already differ for both compilers.

As normally each ATL transformation is compiled to "ASM" instruction format before it is executed, this load-time superimposition approach significantly improves scalability. Only the ATL modules that have changed need to be recompiled, regardless of the other ATL modules it will be combined with. Consider the first scenario given in subsection 4.1: by separating the general copying functionality (UML2Copy module) from the specific refinement functionality (UML2Profiles module), we have achieved better scalability in our development process where we don't have to recompile ±200 copying rules each time we change a refinement rule.

The cost of the improved scalability is the dependency on the ATL compiler implementation: changes to the ATL compiler may violate the assumptions made by the superimposition implementation and effectively break it. Our experience so far shows that ATL compiler changes are not very common and typically concern small changes. We have managed to adapt our implementation in a matter of hours when ATL switched from the `atl2004` compiler to the `atl2006` compiler. The dependency on the output of the ATL compiler also causes some limitations that cannot easily be addressed. By inlining super-rule instructions into the sub-rule operations in rule inheritance, for example, it becomes impossible to do any manipulation of super-rules after compilation. Such limitations require the compiler to be modified.

The performance overhead of the superimposition itself is minimal. The ATL engine already keeps an internal look-up table of available operations when loading a transformation module. Module superimposition updates that table as new modules are loaded on top of the previously loaded modules. The adaptation of the `main`, `__matcher__` and `__exec__` operations form a very small overhead, as only a small fraction of the instructions have to be analysed and manipulated. As a comparison, parsing ASM files currently takes much longer than performing superimposition.

7 Superimposition of QVT Relations transformations

Module superimposition can be used for other languages than ATL, as long as the transformation language supports a number of required features/semantics. The following is a list of language requirements for the application of module superimposition:.

Labeled rules and/or functions – module superimposition operates at the level of matched, lazy and called rules, and helper methods and attributes in the case of ATL. It uses the name – or *label* – of a rule, and the signature of a helper method, to determine whether a rule/method/attribute is *appended* to the base module or *overridden*. Therefore, a transformation language must at least have labels for its rules, functions or fields, such that module superimposition can distinguish between *append* and *override* situations. For functions, these labels can also take the form of signatures. It is also assumed here that one can distinguish between the different kinds of constructs (rules/functions/fields) for the purpose overriding elements of the same kind only.

Replaceable rule and/or function construct – module superimposition can append and replace rules, methods and attributes in the case of ATL. This is possible, because rules, methods and attributes can be replaced by compatible rules, methods and attributes, as explained in section 3 in terms of safe rule overriding. In order to apply module superimposition to a transformation language, it must be able to append new rules, functions or fields, and replace existing ones. Determining when it is *safe* to append or replace a rule/function/field relies heavily on the transformation language's semantics, and must be re-evaluated for each transformation language.

Module construct – module superimposition composes one *module* of rules, methods and attributes with another *module* of rules, methods and attributes in the case of ATL. A transformation language must have a *module* construct that separates the group of base rules/functions/fields from the superimposed rules/functions/fields.

QVT Relations is a transformation language that satisfies these requirements. In the Relations language, a transformation between models is specified as a set of labeled relations that must hold for the transformation to be successful. Each model in the transformation conforms to a model type, which is a specification of the kind of model elements that can occur in a conforming model. A model type is typically represented by a metamodel. The models in a transformation are named and are bound to a specific model type.

As ATL historically served as a submission to the QVT Request For Proposals [15], QVT Relations shows similarities to ATL: QVT Relations uses the term "transformation" for a transformation module and "relation" for a transformation rule, where *top* relations correspond to the automatically invoked ATL matched rules, and normal relations correspond to ATL lazy rules. There are also some important differences between QVT Relations and ATL: QVT Relations transformations can be bidirectional and QVT Relations is strictly declarative, while ATL is a hybrid declarative/imperative language. Furthermore, QVT Relations uses the notion of *checking* transformations and *enforcing* transformations, where a checking transformation only verifies if the relations hold, and an enforcing transformation actually changes the output model such that the relations hold. Whereas ATL matched/lazy rules make a clear distinction between **from** and **to** parts, QVT relations can have two or more *domains*. These domains can be used as a matching expression, or **from** part, as well as an output expression, or **to** part. Domains can be marked as **checkonly** or **enforce**, which allows one to indicate whether a domain is allowed to be used for creating output. Each relation can have a **when** and **where** clause. A **when** clause specifies a necessary condition under which a relation applies or is enforced. A **where** clause specifies additional conditions that apply or are enforced whenever the relation containing the **where** clause applies or is enforced. It is possible to specify OCL constraints as well as other relations in **when** and **where** clauses.

```
transformation UML2Copy
(IN:UML2, OUT:UML2) {
  top relation Model {
    enforce domain IN s: uml::Model {
      name = n,
      visibility = v,
      viewpoint = vp,
      profileApplication = pa
    };
    enforce domain OUT t: uml::Model {
      name = n,
      visibility = v,
      viewpoint = vp,
      profileApplication = pa
    };
  }
  ...
}
```

Listing 13 UML2Copy QVTR transformation

Listing 13 shows our UML2Copy example from Listing 1 as a Relations specification, with the addition of the "profileApplication" property assignment. The dots indicate that we have omitted a number of relations. We still need one relation for each meta-class in the UML2 meta-model, just like in our ATL version of the transformation described in subsection 4.1. Each of these relations follow the same pattern as the given "Model" relation. Now let's consider the same scenario, in which we superimpose the Relations version of UML2Profiles on UML2Copy. Listing 14 shows UML2Profiles as defined in QVT Relations.

The UML2Profiles transformation definition uses a **checkonly** domain to find the "Accessors" profile element, such that it can be applied to the model, if necessary. Both relations in UML2Profiles use a **when** clause to define the condition under which the relation should

```
transformation UML2Profiles
(IN:UML2, ACCESSORS:UML2, OUT:UML2) {
  top relation Model {
    enforce domain IN s: uml::Model {
      name = n,
      visibility = v,
      viewpoint = vp,
      profileApplication = p
    };
    enforce domain OUT t: uml::Model {
      name = n,
      visibility = v,
      viewpoint = vp,
      profileApplication = p
    };
    checkonly domain ACCESSORS
    accessorsProfile: uml::Profile {
      name = 'Accessors'};
    when {p->select(a|
      a.appliedProfile=accessorsProfile)
      ->notEmpty();
    }
  }
  top relation ModelProfile {
    enforce domain IN s: uml::Model {
      name = n,
      visibility = v,
      viewpoint = vp,
      profileApplication = p
    };
    enforce domain OUT t: uml::Model {
      name = n,
      visibility = v,
      viewpoint = vp,
      profileApplication = p->union(Set{
        pa: uml::ProfileApplication {
          applyingPackage = t,
          appliedProfile =
            accessorsProfile}})
    };
    checkonly domain ACCESSORS
    accessorsProfile: uml::Profile {
      name = 'Accessors'
    };
    when {
      p->select(a|
        a.appliedProfile=accessorsProfile)
      ->isEmpty();
    }
  }
}
```

Listing 14 UML2Profiles QVTR transformation

be applied (i.e. enforced). The "Model" relation applies when the "Accessors" profile has already been applied to the source model, and the "ModelProfile" relation applies if this is not the case.

The illustration of superimposition in Fig. 4 is also valid for QVT Relations. A QVT *transformation* is the equivalent of an ATL *module* and a QVT *relation* is the equivalent of an ATL *rule* for the purpose of superimposition. When the two relations "Model" and "ModelProfile" are superimposed on UML2Copy, the "Model" relation is overridden and the "ModelProfile" relation is added to the base transformation.

The fact that relations can be bidirectional has minor influence on superimposition semantics. Bidirectional relations are only interpreted in one direction at a time and can be seen as two transformation rules expressed in a single relation. Since module superimposition operates at the granularity of relations, this means that one can only extend and override a whole relation expression, including all directions in which it can be interpreted.

The effect of overriding or adding relations to the existing set of relations is not so easy to specify, unfortunately. Listing 15 shows an attempt at expressing the UML2ExtendedCopy ATL module from Listing 2 in QVT Relations. QVT's **when** clause is used to achieve the same effect as rule inheritance in ATL: the "Model" relation extends the "Package" relation with a binding of the "viewpoint" property (remember that "Model" is a subclass of "Package" – see also Fig. 1). However, when describing the binding of the "packagedElement" property in the "Package" relation, we run into problems. The QVT specification makes no mention of an implicit tracing mechanism for QVT Relations, so we must specify the tracing explicitly. We do this in a **where** clause, in which we state that "spe" is related to "tpe" via the "PackageableElement" relation. However, the "PackageableElement" relation does not apply to sequences of "PackageableElement", but single "PackageableElement" instances. It is not clear whether QVT Relations can perform implicit "mapping" of the "PackageableElement" relation to each of the values in the "spe" and "tpe" sequences.

Assuming that the relationship between "spe" and "tpe" can be expressed in terms of the other relations, the effect on module superimposition is clear. Each reference to a relation requires any overriding relations to be safe substitutes for the overridden relation. A safe substitute relation must have at least the same number of domains available, with the exact same types. This is in contrast to the notion of safe rule overriding in ATL, where the **from** part must match a superset of elements (the same or more general element type) and the **to** part must create elements that are of the same or more specific type. This is because QVT relations can be interpreted in multiple directions and any domain can play the role of both an ATL **from** and **to** part. An exception to this situation are the **checkonly** domains, which only play the role of an ATL **from** part. As such, the type of a domain that overrides a **checkonly** domain must be the same or more general than the original type (*covariant*). Finally, it is possible to override a **checkonly** domain with an **enforce** domain, but not the other way around. A **checkonly** domain can only be used as input, while an **enforce** domain can be used as both input and output. As such, **enforce** implies **checkonly**, but not the other way around.

In order to provide semantics for QVT Relations superimposition, we can write a higher-order transformation definition that performs superimposition on two

```
transformation UML2ExtendedCopy
(IN:UML2, OUT:UML2) {
  top relation Model {
    enforce domain IN s: uml::Model {
      viewpoint = vp,
    };
    enforce domain OUT t: uml::Model {
      viewpoint = vp,
    };
    when {
      Package(s, t);
    }
  }
  top relation Package {
    enforce domain IN s: uml::Package {
      packagedElement = spe,
      profileApplication = pa
    };
    enforce domain OUT t: uml::Package {
      packagedElement = tpe,
      profileApplication = pa
    };
    when {
      PackageableElement(s, t);
    }
    where {
      PackageableElement(spe, tpe);
    }
  }
  top relation PackageableElement {
    enforce domain IN
    s: uml::PackageableElement {};
    enforce domain OUT
    t: uml::PackageableElement {};
    when {
      NamedElement(s, t);
    }
  }
  top relation NamedElement {
    enforce domain IN s: uml::Package {
      name = n,
      visibility = v
    };
    enforce domain OUT t: uml::Package {
      name = n,
      visibility = v
    };
  }
}
```

Listing 15 UML2Copy QVTR transformation

QVT Relations transformation definitions. Such a higher-order transformation definition is preferably expressed in QVT Relations as well, as the semantics of module superimposition for QVT Relations should be based only on the semantics of QVT Relations itself. However, if we want the same value as for the ATL superimposition semantics, we should provide executable semantics. That requires a tool that implements QVT Relations.

There are a number of tools that implement QVT Relations, such as Medini QVT[7], MOMENT-QVT [2],

[7] http://projects.ikv.de/qvt

ModelMorf[8] and Eclipse M2M Declarative QVT[9]. Medini QVT claims to provide a full implementation of QVT Relations, based on its textual concrete syntax. It does not make available QVT's meta-model to the end user and it is therefore not possible to use Medini QVT for higher-order transformations. The Medini QVT tool has also been used to verify the syntax of the QVT Relations transformation definitions in this paper. On closer inspection of the Medini QVT source code bundle, we found that the Medini developers have discovered issues with the QVT Relations standard that made it impossible for them to implement all of it[10]. Then again, the problem with the binding of the "packagedElement" property described in Listing 15 does not occur in Medini QVT. Medini QVT implicitly binds all containment properties in a relation, which includes "packagedElement". As a result, the "packagedElement" binding and the accompanying **where** clause do not appear in the Medini QVT version of UML2ExtendedCopy. This implicit binding is performed according to an implicit tracing mechanism, which translates elements in one model to elements in another model using the available relations. MOMENT-QVT does appear to use the QVT meta-model, but does not provide a download link and claims to have only a partial implementation of QVT Relations. ModelMorf is a commercial tool that is currently in beta stage and provides a partial implementation of QVT Relations. Eclipse M2M Declarative QVT is currently also under development and does not yet provide a running tool. It does provide a QVT meta-model and editor. It's engine is built around the ATL virtual machine.

As none of the tools implementing QVT Relations are fully functional yet, we cannot currently provide executable semantics for QVT Relations superimposition. According to the Medini QVT developers, QVT Relations itself is not even well-defined yet. It also does not make sense to implement QVT Relations superimposition when QVT Relations itself is not yet adequately implemented. The Eclipse M2M Declarative QVT tool looks most promising at this moment, as the ATL engine will be used as its execution platform. Therefore, an implementation of QVT Relations superimposition may be derived from the ATL superimposition implementation for the Eclipse M2M Declarative QVT tool.

8 Related work

In this section, we provide a comparison with the composition mechanisms available in several other transformation languages. Table 1 provides an overview of this comparison. The remainder of this section discusses each of the transformation languages mentioned in detail.

In the domain of model transformation languages, (internal) composition techniques are relatively new. In the domain of graph transformation [13], *critical pair analysis* [14] can be used to analyse which transformation rules can be used together. Critical pair analysis can determine in a pairwise way how graph transformation rules interact. This interaction can be a lot more complex than it is in ATL, as graph transformations are performed in-place, rules can trigger on each other's output, and rules can be triggered multiple times as long as there is matching input. Critical pair analysis detects possible conflicts between graph transformation rules. A conflict indicates that a pair of rules cannot be applied in parallel, as applying them in different order yields different results. A composition of graph transformation rules applied in parallel most closely resembles an ATL transformation module. As graph transformations are in-place transformations without implicit tracing mechanism [5], there is no difference between applying one graph transformation rule after the other or applying them "together" like rules in an ATL module, as long as there are no critical pairs among the rules. As there is no separate module construct in graph transformation, there is no module composition construct either.

The Epsilon Transformation Language (ETL) [11] uses transformation *strategies* to specify the default behaviour for elements that don't match against any transformation rule. Strategies are defined in Java as engine plug-ins instead of in the ETL itself, whereas with module superimposition, such default behaviour can be defined directly in the transformation language. ETL strategies provide opportunity for performance optimisation at the cost of additional complexity for the end user of the transformation language. ETL transformation executions can be run in sequence, while preserving tracing information, using ETL workflows. This composition mechanism closely resembles RubyTL's phasing mechanism, which is discussed later in this section. ETL also has a module construct, just like ATL, with the difference that ETL modules can import other modules (ATL modules can only import libraries with helpers). ETL module import makes all rules from the imported module available to the importing module. The importing module can override any of those rules by specifying a new rule with the same name as the overridden rule. This is similar to module superimposition, where rules of different modules are combined and can also be overridden by name. The key difference lies in the fact that ETL module import is specified at design-time inside the module itself and that multiple module imports can be specified in the same module, which may require conflict resolution in case multiple imported modules include candidates of the same rule. Module superimposition, on the other hand, is specified per run configuration and there is a strict ordering of superimposed modules. Another differ-

[8] http://www.tcs-trddc.com/ModelMorf/
[9] http://wiki.eclipse.org/M2M/Relational_QVT_Language_(QVTR)
[10] http://tinyurl.com/openproblems-txt

Language	Composition mechanisms	Key advantage	Key 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 refine existing rules	Tight coupling and possible overriding conflicts
QVT OM	Inheritance, access, extends	Reuse as well as refine 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

Table 1 Comparison overview of composition mechanisms in transformation languages.

ence is that module superimposition does not imply an explicit import of underlying modules in the superimposed modules. There is no direct (lexical) dependency between overridden rules and their overriding rules. Only if a module explicitly invokes lazy/called rules from another module, an explicit import is required. This also means that overriding rules cannot refer to and reuse elements from their overridden rules. In ATL, this behaviour is the responsibility of rule inheritance, which involves an explicit (lexical) reference to the inherited rule. ATL rule inheritance is currently limited to single modules, but can conceptually be combined with module superimposition. The semantics of this combination can be derived from section 5. ETL also supports rule inheritance via the *extends* keyword, but no further information on the semantics of this keyword is available. Finally, as module superimposition is defined per run configuration instead of inside the various modules, module superimposition allows for changing the composition of modules without changing the modules themselves.

The QVT Operational Mappings language [16] uses the **access** and **extends** operators to compose transformation definitions. The Operational Mappings language differs from QVT Relations and ATL in that it does not have the concept of matching, but uses an explicit "`main()`" operation instead. QVT Operational Mappings has an implicit tracing mechanism, similar to the one that ATL uses. Operational Mappings uses mapping operations that can be invoked on model elements instead of rules that match against model elements. Mapping operations can inherit the behaviour of other mapping operations, which has an effect that is similar to ATL rule inheritance. An inheriting mapping operation performs all assignments of its super operation, plus a number of extra assignments. The **access** composition operator imports another transformation definition that can be accessed as if it were an object-oriented class: it can be instantiated for an input model and its `main()` can be triggered. The **extends** composition operator also imports another transformation definition, but it places all the imported mapping operations in the current namespace, such that they can be invoked as if they were part of the current transformation definition. It is not clear what happens if the imported transformation definition contains a mapping operation with a signature that already occurs in the current transformation definition. If mapping operations with the same signature in the extending transformation definition replace the operations in the extended transformation definition, then the semantics are similar to ETL's module import construct and QVT Operational Mappings **extends** is comparable to module superimposition. The same observations apply as for ETL module import versus module superimposition.

RubyTL is another rule-based transformation language [4], which also supports a composition mechanism that works on sets of rules. The mechanism is called the *phasing* mechanism. By grouping rules together in a *phase*, the execution order of the rules is defined. Grouping rules in phases has a similar effect as putting them in separate transformation definitions: rules can trigger on the output of the rules in a previous phase. Therefore, the phasing mechanism essentially implements an external composition mechanism (chaining of transformation executions) as an internal mechanism. RubyTL supports another composition mechanism in the form of *refinement* rules. A refinement rule specifies additional assignments on top of any existing rules, and behaves like an implicit form of ATL's rule inheritance. Refinement rules use implicit tracing information to find their **from** and **to** elements, and then perform the additional assignments. Refinement rules cannot override any previous assignments made by the regular rules that they refine. Refinement rules are therefore inherently safe, whereas rule overriding in module superimposition can be unsafe. The scenario in subsection 4.3 can be implemented by prepending the "superimposed" phase to the "base" phase in the execution order. Prepending the "superimposed" phase allows for a kind of overriding behaviour, where the "overridden" rule's trigger condition no longer occurs after the "overriding" rule has been applied (comparable to critical pairs in graph transformation). We assume here that the creation of one output model can be spread over multiple phases. It is not clear whether the implicit tracing information of a preceding phase is available in subsequent phases. If this is the case, then it is possible to implement the scenario described in subsection 4.1 in a similar way, except that the "superimposed" phase must be *appended* to the "base" phase, and only refinement rules are used in the "superimposed" phase.

The latter approach can only be used to perform strict refinement, and no base behaviour can be overridden. It can be difficult to understand the overriding behaviour of the phasing mechanism, as one has to keep track of which rule matches which elements.

In the domain of program transformation, the Conditional Transformations approach (CTs) has a special composition mechanism for combining multiple transformations into one transformation [10]. The CT approach is similar to graph transformations in that each transformation consists of one rule. Instead of negative application conditions, CTs use logic *conditions* to narrow down the scope of elements on which the rule triggers. The CT composition mechanism allows for the composition of multiple CT rules. The result is a transformation with multiple rules, not unlike ATL or QVT Relations. The composed CT is a *sequence* of rules, where the rule sequence may be an AND-sequence or an OR-sequence. The AND and OR refer to the trigger condition of the rules: in an AND-sequence, all rule conditions must hold for the transformation to be executed. In an OR-sequence, individual rules may trigger while others do not. CT composition achieves the same goal as module superimposition, since it can combine pre-existing transformation rules in any way. Because the nature of CT rules is very different from ATL rules, the composition mechanisms are different as well. CT rules are independently defined and may be applied in sequence, while ATL rules are defined in combination and interact via the implicit tracing mechanism. The same observations as for graph transformation apply here.

9 Future work

We intend to investigate more use cases of module superimposition in the future. A candidate use case we are currently looking into is the leverage of ATL's implicit tracing mechanism to automatically "update" models that refer to the model being transformed. When a source model is transformed by ATL, the result is stored in a separate target model. There are situations in which other models contain references to this source model. On many occasions, these references should be updated to point to the target model instead. When using module superimposition, one can superimpose copy transformation modules for each model that references the source model. The copy transformation normally results in exactly the same model, but when it is superimposed, ATL's implicit tracing mechanism interacts with the copy transformation, such that all references to "mapped" model elements from the source model are translated to their target model counterparts as soon as the reference is assigned. This allows us to implicitly translate all references to the source model to references to the target model.

The current implementation of ATL rule inheritance is based on inlining of the super-rule into the sub-rule. In the future, this implementation may be changed to use dynamic look-up of super-rules after a transformation module has been compiled. This allows superimposition of super-rules as well as rule inheritance across superimposed modules, since all super- and sub-rules are still available at load-time. Dynamic look-up of super-rules also alleviates the limitation of rule overriding in module superimposition, where an overriding rule cannot reuse behaviour of the overridden rule. Instead of overriding a rule, one could add a new sub-rule in the superimposed module that reuses and extends the behaviour of the super-rule in the base module.

The current state of the QVT Relations tools did not allow us to provide executable semantics for QVT Relations superimposition. As soon as a fully functional QVT Relations engine with support for higher-order transformation becomes available, we intend to work on an executable semantics for QVT Relations superimposition. It is not expected that such an executable semantics can be applied to all QVT Relations implementations, as the official specification is not well-defined. This is illustrated by the developers of the Medini QVT tool in section 7, who have indicated that they needed to make certain assumptions about the semantics where the specification gave no information. The most practical solution to this problem is for the Object Management Group to provide a reference implementation for their QVT standard, as is commonly done for computing standards. An executable semantics for module superimposition based on a reference implementation is by definition transferable to other implementations.

The Eclipse M2M Declarative QVT tool mentioned in section 7 uses the ATL virtual machine as its execution platform. Our module superimposition implementation for ATL operates on the bytecode that goes into the ATL virtual machine. This makes it easier to port the implementation of ATL module superimposition to the Eclipse M2M Declarative QVT tool, as it uses the same bytecode format. We plan to follow the development of the Eclipse M2M Declarative QVT tool closely and investigate when it is possible to port our implementation of module superimposition for this tool.

10 Conclusion

This paper has presented an approach for *internal* composition of model transformations written in a rule-based model transformation language. By a rule-based model transformation language, we mean a model transformation language that has the concept of *modules* containing a number of rules. Our composition approach, called *module superimposition*, allows for the composition of two or more transformation modules in one single transformation execution. It therefore allows one to split up a model transformation into multiple, reusable and maintainable transformation modules that can later be composed. Module superimposition is implemented for ATL, but is also applicable to the QVT Relations language. Module superimposition has been applied in our MDE case study[11] on UML 2.x refinement transformations, Java API model to platform ontology transformations [19] as well as the build script generators for our case study's configuration language.

One main use case of module superimposition is to achieve a *base behaviour* from the transformation engine. By default, ATL does not transform anything in the input models and will give back an empty output model. For refinement or refactoring transformations, most elements should simply be copied and only a few elements are modified. In ATL, this means that every refinement/refactoring transformation consists mostly of copying rules. ATL refining mode has been introduced to tackle this issue, but it cannot deal with customised copying requirements. Module superimposition allows one to modularise all copying rules into a separate copying transformation. That copying transformation may include any special conditions that can be expressed in ATL. By separating the base behaviour from the specific behaviour, we achieve better maintainability through reduced code duplication in the transformation modules. Finally, reusability is improved by the ability to extend and adapt general transformation modules.

As module superimposition is a load-time composition technique, operating on the compiled ATL bytecode, it improves ATL's scalability. When changing one ATL transformation module, one only has to re-compile that particular module. This means that compiler performance no longer has to degrade with increasing transformation code size, as long as transformation code is separated into multiple transformation modules. The performance overhead of the superimposition itself is minimal. Module superimposition updates the internal ATL rule/helper look-up table as new modules are loaded on top of the previously loaded modules.

Module superimposition works at the granularity of transformation rules in ATL and relations in QVT Relations. It allows one to add new rules/relations and to override existing ones. As ATL already supports decomposition of transformation rules into helpers and called rules, our module superimposition approach can leverage this decomposition. In addition to overriding and adding standard matched rules, it is possible to override and add helpers and called rules as well. Deletion of rules and helpers is not directly supported, but it is possible to replace the trigger condition with a condition that never triggers.

As ATL is currently implemented as a dynamically typed language, the effects of module superimposition on a static type checker haven't been discussed explicitly. However, we have discussed the notion of safe rule overriding in section 3, where we specified the conditions for an overriding rule to be a safe replacement for the overridden rule. This notion of safety basically refers to type safety. As module superimposition in principle allows for unsafe rule overriding, a statically typed transformation language that supports module superimposition must still perform type checking after superimposition. Our higher-order transformation from section 5 specifies the module superimposition semantics in terms of a single, combined transformation module. A module superimposition composition is type correct if and only if this single, combined transformation module is type correct.

ATL supports rule inheritance as another composition mechanism, which can be freely combined with module superimposition in principle. In its current implementation, however, it is not possible to separately superimpose sub- and super-rules in ATL rule inheritance. Only the sub-rules can be manipulated by module superimposition, because the ATL compiler in-lines the super-rules into the sub-rules. This is a limitation that is caused purely by the current ATL compiler implementation and affects only our implementation of module superimposition for ATL. The executable semantics provided in this paper are not affected by this limitation.

Acknowledgements The authors would like to thank Frédéric Jouault for reviewing a draft of this paper and his discussions on the topic. Thanks also go to the ICMT 2008 review committee for providing many helpful comments. Thanks go to the ATL community for trying out module superimposition and helping to iron out any issues.

Appendix A: ATL meta-model

This appendix contains an excerpt of the ATL meta-model that is relevant to the ATL transformation code given in section 5. Fig. 8 shows a graphical representation of the ATL meta-model excerpt. The meta-model is written in the Ecore language[3]. The complete ATL meta-model can be found at http://tinyurl.com/2t5mcp.

Appendix B: Superimpose.atl

This appendix contains the full source code of the Superimpose.atl transformation module discussed in section 5:

```
module Superimpose;

create OUT: ATL from IN: ATL, SUPER: ATL;

helper def: inElements:
Set(ATL!ATL::LocatedElement) =
  ATL!"ATL::LocatedElement"
    .allInstancesFrom('IN')
    ->reject(o|o.isOverridden())->asSet()
  ->union(ATL!"ATL::LocatedElement"
```

[11] http://ssel.vub.ac.be/ssel/research:mdd:casestudies

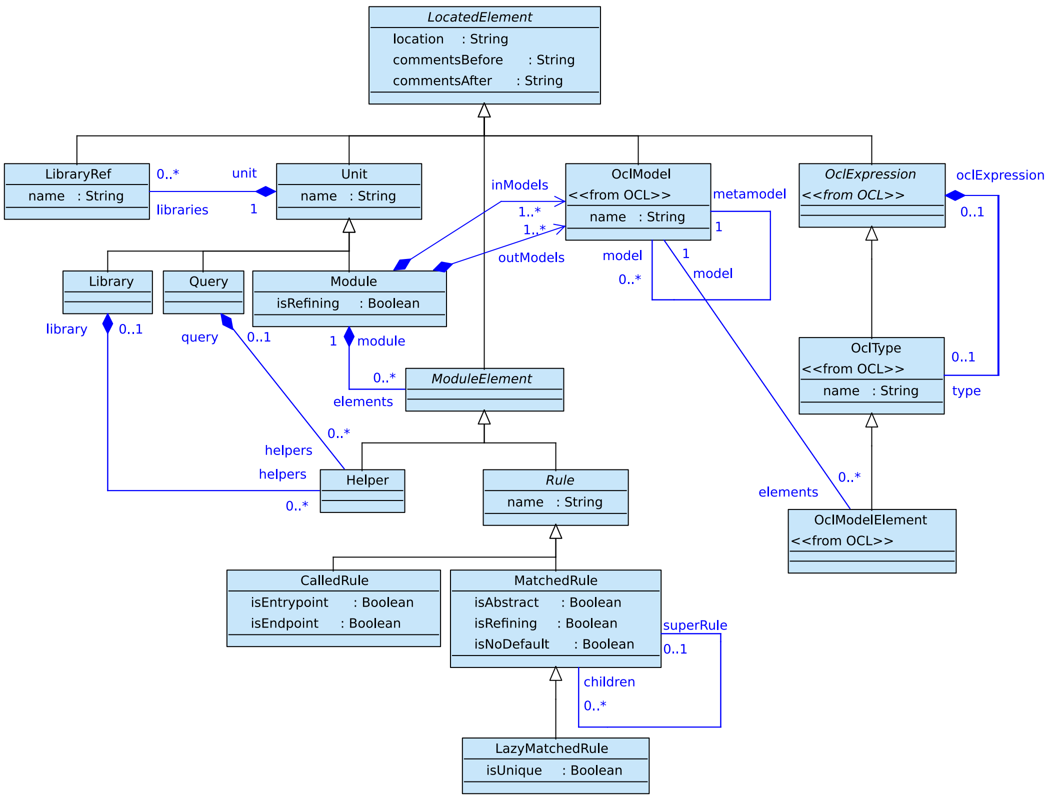


Fig. 8 ATL meta-model excerpt.

```
    .allInstancesFrom('SUPER')
    ->reject(s|
      if s.oclIsKindOf(ATL!"ATL::Rule")
      or s.oclIsKindOf(ATL!"ATL::Helper")
      then s.isOverriding()
      else false endif));

helper def: realInElements:
Set(ATL!"ATL::LocatedElement") =
  ATL!"ATL::LocatedElement"
  .allInstancesFrom('IN');

-- ************ isOverridden ************

helper context ATL!"ATL::LocatedElement"
def: isOverridden(): Boolean =
  let owner: ATL!"ATL::LocatedElement" =
    self.refImmediateComposite() in
  if owner.oclIsUndefined() then false
  else owner.isOverridden() endif;

helper context ATL!"ATL::Rule"
    ATL!"ATL::Rule"
    .allInstancesFrom('SUPER')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());

helper context ATL!"ATL::Helper"
    ATL!"ATL::Helper"
    .allInstancesFrom('SUPER')
    ->exists(r|
      r.definition.feature.name =
        self.definition.feature.name
      and
      r.definition.feature.oclType() =
        self.definition.feature.oclType()
      and
      if r.definition.context_
        .oclIsUndefined()
      or self.definition.context_
        .oclIsUndefined() then
        r.definition.context_ =
          self.definition.context_
      else
        r.definition.context_
            .context_.name =
          self.definition.context_
            .context_.name
      endif);

helper context
ATL!"OCL::VariableDeclaration"
def: isOverridden(): Boolean =
  let owner: ATL!"ATL::LocatedElement" =
    self.refImmediateComposite() in
  let variableExpOverridden: Boolean =
    self.variableExp->exists(e|
      e.isOverridden()) in
  if owner.oclIsUndefined() then
    variableExpOverridden
  else
    owner.isOverridden() or
    variableExpOverridden
  endif;
```

```
helper context ATL!"OCL::OclModel"
def: isOverridden(): Boolean =
    ATL!"OCL::OclModel"
    .allInstancesFrom('SUPER')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());

helper context ATL!"ATL::LibraryRef"
def: isOverridden(): Boolean =
    ATL!"ATL::LibraryRef"
    .allInstancesFrom('SUPER')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());

-- ************ isOverriding ************

helper context ATL!"ATL::LocatedElement"
def :isOverriding(): Boolean =
  let owner: ATL!"ATL::LocatedElement" =
    self.refImmediateComposite() in
  if owner.oclIsUndefined() then false
  else owner.isOverriding() endif;

helper context ATL!"ATL::Rule"
def: isOverriding(): Boolean =
    ATL!"ATL::Rule"
    .allInstancesFrom('IN')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());

helper context ATL!"ATL::Helper"
def: isOverriding(): Boolean =
    ATL!"ATL::Helper"
    .allInstancesFrom('IN')
    ->exists(r|
      r.definition.feature.name =
        self.definition.feature.name
      and
      r.definition.feature.oclType() =
        self.definition.feature.oclType()
      and
      if r.definition.context_
        .oclIsUndefined()
      or self.definition.context_
        .oclIsUndefined() then
        r.definition.context_ =
          self.definition.context_
      else
        r.definition.context_
            .context_.name =
          self.definition.context_
            .context_.name
      endif);

helper context ATL!"OCL::OclModel"
def: isOverriding(): Boolean =
    ATL!"OCL::OclModel"
    .allInstancesFrom('IN')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());

helper context ATL!"ATL::LibraryRef"
def: isOverriding(): Boolean =
    ATL!"ATL::LibraryRef"
    .allInstancesFrom('IN')
    ->exists(r|
      r.name = self.name and
      r.oclType() = self.oclType());

-- ************ overriddenBy ************

helper context ATL!"ATL::Rule"
def: overriddenBy(): ATL!"ATL::Rule" =
  let selfInSuper:
  Sequence(ATL!"ATL::Rule") =
    ATL!"ATL::Rule"
    .allInstancesFrom('SUPER')
    ->select(r|
      r.name = self.name and
      r.oclType() = self.oclType()) in
  if selfInSuper->isEmpty() then self
  else selfInSuper->first() endif;

helper context ATL!"ATL::Helper"
def: overriddenBy(): ATL!"ATL::Helper" =
  let selfInSuper:
  Sequence(ATL!"ATL::Helper") =
    ATL!"ATL::Helper"
    .allInstancesFrom('SUPER')
    ->select(r|
      r.definition.feature.name =
        self.definition.feature.name
      and
      r.definition.feature.oclType() =
        self.definition.feature.oclType()
      and
      if r.definition.context_
        .oclIsUndefined()
      or self.definition.context_
        .oclIsUndefined() then
        r.definition.context_ =
          self.definition.context_
      else
        r.definition.context_
            .context_.name =
          self.definition.context_
            .context_.name
      endif) in
  if selfInSuper->isEmpty() then
    self
  else
    selfInSuper->first()
  endif;

helper context ATL!"OCL::OclModel"
def: overriddenBy(): ATL!"OCL::OclModel" =
  let selfInSuper:
  Sequence(ATL!"OCL::OclModel") =
    ATL!"OCL::OclModel"
    .allInstancesFrom('SUPER')
    ->select(r|
      r.name = self.name and
      r.oclType() = self.oclType()) in
  if selfInSuper->isEmpty() then self
  else selfInSuper->first() endif;

-- ************ rules ************

rule Module {
  from s : ATL!"ATL::Module" (
    thisModule.realInElements->includes(s))
  using {
    superElements:
    Sequence(ATL!"ATL::ModuleElement") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.elements
        ->select(e|not e.isOverriding()))
      ->flatten();
    superInModels:
    Sequence(ATL!"OCL::OclModel") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.inModels)
      ->flatten();
    superOutModels:
    Sequence(ATL!"OCL::OclModel") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.outModels)
      ->flatten();
    superLibraryRefs:
    Sequence(ATL!"ATL::LibraryRef") =
      ATL!"ATL::Module"
      .allInstancesFrom('SUPER')
      ->collect(m|m.libraries)
      ->flatten(); }
  to t : ATL!"ATL::Module" (
    location <- s.location,
    commentsBefore <- s.commentsBefore,
    commentsAfter <- s.commentsAfter,
    name <- s.name,
    isRefining <- s.isRefining,
    libraries <- s.libraries
      ->union(superLibraryRefs),
    inModels <- s.inModels
      ->union(superInModels),
    outModels <- s.outModels
      ->union(superOutModels),
    elements <- s.elements
      ->union(superElements))
}

rule OverriddenMatchedRule {
  from s : ATL!"ATL::MatchedRule" (
    if thisModule.realInElements
      ->includes(s) then
      s.oclIsTypeOf(ATL!"ATL::MatchedRule")
      and s.isOverridden()
    else false endif)
  using {
    o : ATL!"ATL::MatchedRule" =
      s.overriddenBy(); }
  to t : ATL!"ATL::MatchedRule" (
    location <- o.location,
    commentsBefore <- o.commentsBefore,
    commentsAfter <- o.commentsAfter,
    name <- o.name,
    isAbstract <- o.isAbstract,
    isRefining <- o.isRefining,
    isNoDefault <- o.isNoDefault,
    outPattern <- o.outPattern,
    actionBlock <- o.actionBlock,
    variables <- o.variables,
    inPattern <- o.inPattern,
    children <- o.children,
    superRule <- o.superRule)
}

rule OverriddenLazyMatchedRule {
  from s : ATL!"ATL::LazyMatchedRule" (
    if thisModule.realInElements
      ->includes(s) then
      s.isOverridden()
    else false endif)
  using {
    o : ATL!"ATL::LazyMatchedRule" =
      s.overriddenBy(); }
  to t : ATL!"ATL::LazyMatchedRule" (
    location <- o.location,
    commentsBefore <- o.commentsBefore,
    commentsAfter <- o.commentsAfter,
    name <- o.name,
    isAbstract <- o.isAbstract,
    isRefining <- o.isRefining,
    isNoDefault <- o.isNoDefault,
    isUnique <- o.isUnique,
    outPattern <- o.outPattern,
    actionBlock <- o.actionBlock,
    variables <- o.variables,
    inPattern <- o.inPattern,
    children <- o.children,
    superRule <- o.superRule)
}

rule OverriddenCalledRule {
  from s : ATL!"ATL::CalledRule" (
    if thisModule.realInElements
      ->includes(s) then
      s.isOverridden()
    else false endif)
  using {
    o : ATL!"ATL::CalledRule" =
      s.overriddenBy(); }
  to t : ATL!"ATL::CalledRule" (
    location <- o.location,
    commentsBefore <- o.commentsBefore,
    commentsAfter <- o.commentsAfter,
    name <- o.name,
    isEntrypoint <- o.isEntrypoint,
    isEndpoint <- o.isEndpoint,
    outPattern <- o.outPattern,
    actionBlock <- o.actionBlock,
    variables <- o.variables,
    parameters <- o.parameters)
}

rule OverriddenHelper {
  from s : ATL!"ATL::Helper" (
    if thisModule.realInElements
      ->includes(s) then
      s.isOverridden()
    else false endif)
  using {
    o : ATL!"ATL::MatchedRule" =
      s.overriddenBy(); }
  to t : ATL!"ATL::Helper" (
    location <- o.location,
    commentsBefore <- o.commentsBefore,
    commentsAfter <- o.commentsAfter,
    definition <- o.definition)
}

rule OclModelElement {
  from s : ATL!"OCL::OclModelElement" (
    thisModule.inElements->includes(s))
  to t : ATL!"OCL::OclModelElement" (
    location <- s.location,
    commentsBefore <- s.commentsBefore,
    commentsAfter <- s.commentsAfter,
    name <- s.name,
    type <- s.type,
    model <- s.model.overriddenBy())
}
```

References

1. ATLAS group, LINA and INRIA, Nantes, France: Specification of the ATL Virtual Machine (2005). URL http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification%5Bv00.01%5D.pdf. Version 0.1
2. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic specification of a model transformation engine. In: L. Baresi, R. Heckel (eds.) Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE'06), Vienna, Austria, *Lecture Notes in Computer Science*, vol. 3922, pp. 262–277. Springer-Verlag (2006)
3. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse Series. Addison Wesley Professional (2003)
4. Cuadrado, J.S., Molina, J.G.: Approaches for model transformation reuse: Factorization and composition. In: Vallecillo et al. [17], pp. 168–182
5. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3), 621–645 (2006)
6. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (2005)
7. Jouault, F., Kurtev, I.: On the architectural alignment of atl and qvt. In: Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006), Dijon, France (2006)
8. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture : Practice and Promise. Addison Wesley Professional (2003)
9. Kleppe, A.G.: First european workshop on composition of model transformations - CMT 2006. Technical Report TR-CTIT-06-34, Enschede (2006)
10. Kniesel, G., Koch, H.: Static composition of refactorings. Science of Computer Programming **52**(1-3), 9–51 (2004). Special issue on Program Transformation
11. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: Vallecillo et al. [17], pp. 46–60
12. Kurtev, I., van den Berg, K., Jouault, F.: Evaluation of rule-based modularization in model transformation languages illustrated with ATL. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, pp. 1202–1209. ACM Press, New York, NY, USA (2006)
13. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
14. Mens, T., Taentzer, G., Runge, O.: Detecting structural refactoring conflicts using critical pair analysis. Electr. Notes Theor. Comput. Sci. **127**(3), 113–128 (2005)
15. Object Management Group, Inc.: Request for Proposal: MOF 2.0 Query / Views / Transformations RFP (2004). Ad/2002-04-10
16. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2008). Version 1.0, formal/08-04-03
17. Vallecillo, A., Gray, J., Pierantonio, A. (eds.): First International Conference on Model Transformation (ICMT 2008), *Lecture Notes in Computer Science*, vol. 5063. Springer-Verlag (2008)
18. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In: Vallecillo et al. [17]
19. Wagelaar, D., Van Der Straeten, R.: Platform ontologies for the model-driven architecture. European Journal of Information Systems **16**(4), 362–373 (2007)

Dennis Wagelaar is a post-doctoral researcher at the System and Software Engineering Lab of the Vrije Universiteit Brussel in Belgium. He received a Ph.D. in Science at the Vrije Universiteit Brussel for his dissertation "Platform Ontologies for the Model Driven Architecture" and holds a MSc. degree in Computer Science from the University of Twente (The Netherlands). His research interests are model-driven engineering, model transformation and using knowledge-based techniques in the field of software engineering.

Ragnhild Van Der Straeten received the degrees of Licentiate in Applied Mathematics at the Universiteit Gent (Belgium), and Licentiate in Applied Computer Science and Ph.D. in Science at the Vrije Universiteit Brussel (Belgium). Currently, she is a post-doctoral researcher at the System and Software Engineering Lab at the Vrije Universiteit Brussel. She has published many peer-reviewed international articles on the topic of inconsistency management in model-driven software engineering. She has been co-organiser and program committee member of international workshops and conferences. She is a member of the ACM.

Dirk Deridder is a post-doctoral researcher at the System and Software Engineering Lab of the Vrije Universiteit Brussel, Belgium. His Ph.D. dissertation entitled "A Concept-Centric Environment for Software Evolution in an Agile Context" was situated in the field of meta programming/modelling, software evolution/malleability, and model/code co-evolution. Since 1997 he has been active in several research projects with industry covering a wide range of topics including compliance checking, software factories, domain engineering, interactive media development, software generation, software reengineering, software variability, and workflow modelling. Currently he is one of the coordinators of the interuniversity MoVES network which focuses on fundamental issues in software modelling, verification and evolution.