Composition Techniques for Rule-based Model Transformation Languages

Dennis Wagelaar*

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be

Abstract. Model transformation languages have matured to a point where people have started experimenting with model transformation definitions themselves in addition to the language they are written in. In addition to the transformation language properties, the properties of model transformation definitions themselves become important, such as scalability, maintainability and reusability. Composition of model transformations allows for the creation of smaller, maintainable and reusable model transformation definitions that can scale up to a larger model transformation. There are two kinds of composition for model transformations. External composition deals with chaining separate model transformations together by passing models from one transformation to another. Internal composition composes two model transformation definitions into one new model transformation, which typically requires knowledge of the transformation language. This paper focuses on internal composition for two rule-based model transformation languages. One is the ATLAS Transformation Language, which serves as our implementation vehicle. The other is the QVT Relations language, which is a standard transformation language for MOF. We propose a composition technique called module superimposition. We discuss how module superimposition interacts with other composition techniques in ATL, such as helpers, called rules and rule inheritance. Together, these techniques allow for powerful composition of entire transformation modules as well as individual transformation rules. By applying superimposition to QVT Relations, we demonstrate that our composition technique is relevant outside the ATL language as well.

1 Introduction

Model transformations have become increasingly commonplace in model-driven engineering, with a number of stable model transformation languages and tools available. The OMG has even released the MOF Query/View/Transformation standard transformation language [1]. This means that people have started experimenting with model transformations themselves in addition to transformation languages. Whereas the focus initially lay on the expressiveness of transformation languages, other properties are starting to become important, such

* The author's work is part of the VariBru project, which is funded by the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB)

as scalability, maintainability and reusability of model transformation definitions. As model-driven engineering becomes more mature, the model transformation definitions used typically become more elaborate. During the evolution of a model transformation definition, exceptions to the standard transformation scenario are discovered. All these exceptions are then integrated back into the original model 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. Those separate model transformation definitions have be composed in order to achieve the intended transformation result.

Perhaps the most straightforward method of composition is to chain several model transformations together by providing the output of one transformation as input for another transformation. Another method is to compose the rules from a number of transformation definitions into one transformation. The latter method typically requires the model transformations that will be composed to be expressed in the same language. There has been at least one workshop on the topic of model transformation composition [2], where these two methods were labelled as *internal* and *external* transformation composition, respectively. We believe that both composition methods are necessary and complement each other, as we will also demonstrate in this paper.

The focus of this paper lies on internal transformation composition, which means that the composition method 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 two rule-based transformation languages. The first language is the ATLAS transformation language (ATL) [3], which has been used as an implementation vehicle for our experiment. The second language is the QVT Relations [1] standard language, which currently exists as a specification. By translating our composition technique to QVT Relations, we demonstrate that our composition technique is relevant outside the ATL language as well.

The rest of this paper is organised as follows: first, we briefly explain the ATLAS transformation language. After that, we introduce module superimposition for ATL. We discuss module superimposition semantics by means of a higher-order transformation that performs module superimposition. Next, we discuss how module superimposition interacts with other composition techniques in ATL. We will also discuss how module superimposition applies to QVT Relations. We will then discuss related work, followed by the conclusion and future work.

2 ATLAS Transformation Language

The ATLAS Transformation Language (ATL) [3] historically served as a submission to the QVT Request For Proposals [4]. As a consequence, ATL shows simi-

larities to QVT Relations, save some limitations: ATL transformations are unidirectional. Output models are write-only and always start off as empty models. All navigation in ATL is done on read-only input models. QVT checking transformations are typically implemented as ATL *queries*, while enforcing transformations are represented in ATL as *modules*. Furthermore, ATL is a *hybrid* language, providing declarative as well as imperative language constructs. For the purpose of explaining our composition technique, we will only discuss ATL transformation modules.

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. These compare to QVT *top-level* relations and *non-top-level* relations in that matched rules are automatically triggered, while called rules must be invoked from a matched rule. Listing 1.1 shows an example ATL module that copies a UML Model element to another UML Model element.

```
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 1.1. UML2Copy transformation module

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 at run-time. 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 transformation module has one *matched* rule named “Model”. Since ATL transformations are unidirectional, ATL rules don’t have a **domain** construct like QVT relations. Instead, ATL 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 Model copy has the same name, visibility and viewpoint values

as the original Model instance. Listing 1.2 shows how multiple matched rules interact.

```
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 1.2. UML2ExtendedCopy transformation module

The “Model” rule now includes an assignment of the “packagedElement” property. The “packagedElement” property 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, the target element “t” of the “Model” rule is supposed to contain the target “packagedElement” elements, just like the source element “s” contains the source “packagedElement” elements. ATL automatically translates assignments of source elements to their target element counterparts whenever those source elements trigger a matched rule in the transformation module.

This kind of source-to-target element tracing [5] is defined by the **from** element and the first **to** element[1]. This tracing information is used to translate an assignment of source elements to target elements: the target “packagedElement” collection in the “Model” rule will not contain the elements of “s.packagedElement”, but rather the 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”.

---

[1] Tracing information for the other **to** elements is also recorded, but must be retrieved explicitly in ATL via an API call.

3 Module superimposition

While ATL transformation modules are normally run by themselves, that is one transformation module at a time, it is also possible to *superimpose* several transformation modules on top of each other. The end result is a transformation module that contains the union of all transformation rules. It is also possible for a transformation module to *override* rules from the transformation modules it is superimposed upon. Rule overriding is done by *name*: superimposed rules with the same name as an existing rule override the existing rule. This allows for rule-level adaptation of one transformation module by another and improves reusability of transformation modules.

Fig. 1 shows an example of a typical use case for superimposition: 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 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 meta-model. 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 UML2Profiles module is superimposed on the UML2Copy module. It overrides the "Model" rule, which copies each "uml::Model" instance, by a version that checks that 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. 1 that are not ~~struck out~~.

ATL has a number of other constructs besides matched rules, such as lazy rules, called rules, helper attributes and helper methods. Similar 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. Also note that each named ATL construct has its own distinct namespace in ATL, such that name clashes between rules and helpers, for example, are avoided. As such, it is impossible to override a rule by a helper with module superimposition.

Note that superimposition is a load-time construct: there is no real transformation module that represents the result of superimposing several modules on top of each other. Instead, several modules are simply *loaded* on top of each other, overriding existing rules and adding new rules. As normally each ATL

[2] http://ssel.vub.ac.be/viewvc/UML2CaseStudies/uml2cs-transformations/UML2Copy.atl?revision=7380&view=markup

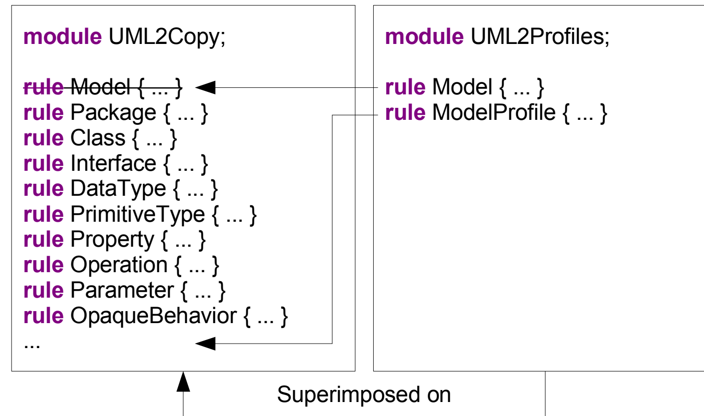

Fig. 1. ATL superimposition example.

transformation is compiled to ASM 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. The performance overhead of the superimposition itself is minimal. The ATL engine already keeps an internal look-up table of available rules and helpers when loading a transformation module. Module superimposition simply updates that table as new modules are loaded on top of the previously loaded modules. In addition, it updates the "main" procedure of the first loaded transformation to include any new rules/helpers for every superimposed transformation module.

3.1 Usage scenarios

Module superimposition is a good way to achieve a specific "base behaviour" of the transformation engine, such as copying the input model to the output model. Superimposition can deal with non-standard situations, such as having multiple input (and/or output) models. Our example UML2Copy transformation is meant to only copy elements from the model "IN" to the model "OUT". Listing 1.3 shows the UML2Profiles transformation module that is superimposed on UML2-Copy.

UML2Profiles adds an extra input model, "ACCESSORS". The "ACCESSORS" model refers to the UML profile that is applied to "OUT". 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".

```
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 1.3. UML2Profiles transformation module

By separating the general copying functionality (UML2Copy) from the specific refinement functionality (UML2Profiles), 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. We have also achieved better maintainability, since it's much easier to find a specific transformation rule within a small, specific transformation module. Maintainability is also improved by reduced code duplication in all available refinement transformation modules; all copying code is now centralised. Finally, reusability is improved by the ability to extend and adapt general transformation modules, such as UML2Copy.

Another usage scenario is the generation of platform ontologies from Java API models expressed in UML [6]. In this scenario, a general transformation module UMLtoAPIOntology.atl is superimposed by either UMLToPackageAPI-

Ontology.atl or UMLToClassAPIOntology.atl to create either a package-level or a class-level ontology of the input Java API model[3].

Yet another scenario is provided by the configuration language of our instant messenger case study[4]. The configuration language meta-model is split up in a general "Transformations" package and a specific "InstantMessenger" package. The ConfigToBuildFile.atl transformation module has also been split up in two parts: one for each meta-model, where ConfigToBuildFile.atl for "InstantMessenger" can be superimposed on ConfigToBuildFile.atl for "Transformations". This allows for reuse of the general "Transformations" infrastructure in other configuration languages and generators.

3.2 Semantics

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. The start of this module is shown in Listing 1.4.

```
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 1.4. Superimpose transformation module

This higher-order transformation module superimposes the "SUPER" transformation module on the "IN" transformation module and writes the result into the "OUT" transformation module[5]. It copies all the elements 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. The transformation rule in Listing 1.5 deals with overridden matched rules.

The "`OverriddenMatchedRule`" transformation rule transforms the overridden matched rule 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

[3] http://ssel.vub.ac.be/ssel/research:mdd:platformkit:ontologies
[4] http://ssel.vub.ac.be/ssel/research:mdd:casestudies
[5] The ATL meta-model can be found at http://tinyurl.com/2t5mcp

```
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 1.5. OverriddenMatchedRule transformation rule

"OUT" as "`s`" did in "IN". This is achieved by ATL's implicit tracing mechanism, which maps all "`s`" references to "`t`" references. This becomes clearer when looking at Listing 1.6, 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, "`elements`" contains the same ordered list, except that its elements are all mapped to their "OUT" counterparts by the ATL tracing mechanism.

The full higher-order transformation is split up in two parts: ATLCopy.atl and Superimpose.atl, where ATLCopy.atl is a simple copying transformation and Superimpose.atl provides the special transformation rules for superimposition[6]. As a proof of concept, Superimpose.atl is superimposed on ATLCopy.atl 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.

3.3 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

[6] http://ssel.vub.ac.be/viewvc/atl-superimposition-semantics/

```
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 1.6. Module transformation rule

called rules in other modules to the modules “below” (i.e. modules that are superimposed upon, not the superimposing modules).

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 decomposition construct called *rule inheritance* [7]. 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. It is currently 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 inlines the super-rules into the sub-rules. No super-rules exist in the ATL bytecode. In addition, it is not possible to inherit from super-rules in another module. This is because module superimposition is performed *after* the compiler does its work. The compiler only operates on a single module at a time.

3.4 Superimposition of QVT Relations transformations

Module superimposition can be used for other languages than ATL, as long as the transformation language has the concepts of rules and modules that contain those rules. QVT Relations is such a transformation language. In the Relations language, a transformation between models is specified as a set of 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 meta-model. The models in a transformation are named and are bound to a specific model type. Listing 1.7 shows our UML2Copy example as a Relations specification.

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

Listing 1.7. UML2Copy QVTR transformation

Note that a special "`inElements`" helper is not necessary. A QVT relation is already defined on the basis of models, not meta-models, so we can explicitly target elements from the model 'IN'. In our example, we only gave a relation for the "uml::Model" meta-class and omitted the other relations. We still need one relation for each meta-class in the UML2 meta-model, just like in our ATL version of the transformation. 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 1.8 shows UML2Profiles as defined in QVT Relations.

The illustration of superimposition in Fig. 1 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.

4 Related work

In the domain of model transformation languages, (internal) composition techniques are relatively new. In graph transformations [8], *negative application conditions* (NACs) are used to inhibit a transformation rule from triggering. A

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

Listing 1.8. UML2Profiles QVTR transformation

NAC essentially is another (partial) transformation rule that is composed with the base transformation rule. It overrides the behaviour of the base transformation rule. As graph transformations are in-place transformations without implicit tracing mechanism [5], there is no difference between applying one rule after the other or applying them "together" like rules in an ATL module[7]. In ATL (and QVT Relations), two or more models are involved and transformation rules interact through implicit tracing. The Epsilon transformation language[8] 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. With module superimposition, such default behaviour can be defined directly in ATL as a normal transformation module.

[7] The order in which rules are applied can be important, however [9]

[8] http://www.eclipse.org/gmt/epsilon/

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*. 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 ATL tracing mechanism.

In the domain of aspect-oriented programming languages, Hyper/J [11] follows an approach similar to superimposition. Hyper/J can merge Java implementations of multiple software dimensions into one Java program. Hyper/J claims to be a symmetric composition approach, in which all dimensions are at the same "level". This is in contrast to superimposition, where one module is superimposed on top of another. Superimposition therefore also allows for overriding. The Composition Filters (CF) [12] approach to aspect-oriented programming includes a "superimposition" language construct. In CF, superimposition refers to *filter modules*, which can be distributed and put on top of object classes. Multiple filter modules can be superimposed on top of each other as well. A filter module can manipulate messages going into or out of objects.

5 Conclusion and future work

This paper has presented an approach for *internal* composition of model transformations written in a rule-based model transformation language. Our composition approach, called *module superimposition*, allows for the composition of two or more transformations into one single transformation. It therefore allows one to split up a model transformation into multiple, reusable and maintainable transformations that can later be composed into one single transformation. Module superimposition is implemented for the ATLAS transformation language, but is also applicable to the QVT Relations language. Module superimposition has been applied in our MDE case study[9] on UML 2.x refinement transformations, Java API model to platform ontology transformations [6] as well as the build script generators for our case study's configuration language.

As module superimposition is a load-time composition technique, operating on the compiled ATL bytecode, it improves ATL's scalability. When changing one

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

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 simply updates the internal ATL rule/helper look-up table as new modules are loaded on top of the previously loaded modules.

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 simply 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. 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.

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.

It is currently 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. In the future, the implementation of ATL rule inheritance can be changed to 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.

There is currently a QVT Relations compiler in development that targets the ATL virtual machine[10]. Module superimposition operates on the bytecode that goes into the ATL virtual machine, which makes it easier to port the implementation of module superimposition to QVT Relations. As soon as the QVT

[10] http://wiki.eclipse.org/M2M/Relational_QVT_Language_%28QVTR%29

Relations compiler is released, an implementation of module superimposition for QVT Relations may become available shortly after.

Acknowledgement

The author would like to thank Frédéric Jouault for reviewing a draft of this paper and his discussions on the topic. Thanks also go to the 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.

References

1. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. (2005) Final Adopted Specification, ptc/05-11-01.
2. Kleppe, A.G.: First European Workshop on Composition of Model Transformations - CMT 2006. Technical Report TR-CTIT-06-34, Enschede (2006)
3. 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)
4. Object Management Group, Inc.: Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. (2004) ad/2002-04-10.
5. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45** (2006) 621–645
6. Wagelaar, D., Van Der Straeten, R.: Platform Ontologies for the Model-Driven Architecture. European Journal of Information Systems **16** (2007) 362–373
7. 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, New York, NY, USA, ACM Press (2006) 1202–1209
8. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electr. Notes Theor. Comput. Sci. **152** (2006) 125–142
9. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci. **127** (2005) 113–128
10. Kniesel, G., Koch, H.: Static Composition of Refactorings. Science of Computer Programming **52** (2004) 9–51 Special issue on Program Transformation.
11. Ossher, H., Tarr, P.: The Shape of Things To Come: Using Multi-Dimensional Separation of Concerns with Hyper/J to (Re)Shape Evolving Software. Comm. ACM **44** (2001) 43–50
12. Bergmans, L., Akşit, M.: Composing Crosscutting Concerns Using Composition Filters. Comm. ACM **44** (2001) 51–57