Towards a Family of Model Transformation Languages

Jesús Sánchez Cuadrado[1]

Universidad Autónoma de Madrid (Spain)
jesus.sanchez.cuadrado@um.es

Abstract. Many model transformation languages of different nature have been proposed during the last years, each one of them suitable for a certain kind of transformation task. However, a complex transformation problem may not fall into a single transformation category, making the solution written in the chosen transformation language suboptimal, as some concerns cannot be handled naturally.
To tackle this issue, we propose to define a model transformation tool as a family of model transformation languages. Each member of the family is a simple language intended to deal with a particular kind of transformation task. In this paper we discuss the different issues involved, such as design decisions, interoperability among languages, and composability. We illustrate the paper with a transformation from UML and OCL to Java, in which languages for pattern matching, mapping, attribution and target-oriented transformations are used. Finally, the approach is validated with a proof-of-concept implementation.

1 Introduction

Model transformation is one of the key elements in Model Driven Engineering (MDE). Hence, in the last years a number of model transformation languages of different nature have been proposed. As acknowledged by the classifications of model transformation languages given in [4] and [13], each language provides a series of features that make it more suitable to address a certain kind of transformation problems. So far, two paths have been taken by transformation language designers: a) keep the language focused or b) add more features to the language in order to widen its scope. The first approach limits the applicability of the language, while the second one tends to pollute the original design.

We have been working on an alternative design, in which a model transformation language is made up of smaller languages. Each language is focused on a specific kind of transformation task, and altogether form a so-called *family of model transformation languages*. In this way, a complex transformation problem could be split into smaller tasks using the most appropriate language for each one of them, with the additional advantage of enhanced declarativeness and intentionality, as languages are really tailored for the problem being solved. Realizing this approach requires a way to make the languages interoperable, as

well as composition mechanisms in order to specify how the results provided by each language contribute to the global transformation result.

This paper reports the initial results of our work building a family of model transformation languages, named Eclectic. We focus on two aspects: the main design decisions which span all languages in the family, and language interoperability and composability mechanisms which are the foundation of our approach. To show the feasibility of the approach the paper is illustrated with a transformation from UML and OCL to Java, which is addressed using four languages of Eclectic, for pattern matching, mapping, attribution and target-oriented transformations. The paper also reports on a proof-of-concept implementation, which includes textual editors and a compiler for the Java Virtual Machine (JVM), that is freely available at [5]. The architecture of the tool is extensible so that it would allow us to integrate domain specific transformation languages (DSTL).

Paper organization. Section 3 explains the main design decisions, presents the running example, and introduces four languages of Eclectic. Section 4 describes the interoperability and composition mechanisms. Section 2 reviews some related work. Section 5 gives some conclusions and outlines the future work.

2 Design of the family

The design of a family of transformation languages must take into account two main concerns: the design of the different languages that build it up, and how to compose them which in turn will require making them interoperable.

In this section we will outline the design principles of Eclectic and illustrate some of its member languages by means of a running example. Section 4 will explain the interoperability and composability mechanisms for them.

2.1 Design principles

The aim of our approach is to tame complex model transformations by promoting intentionality. As a motivating example, ATL is well-known for being a simple, declarative transformation language, but as the transformation problem at hand moves away from being a mapping task, intentionality blurs. This is so because non-declarative constructs such as lazy rules, imperative rules or complex navigation code must be tangled with declarative code. Our design tries to solve this issue by providing separate languages for different kinds of transformation task, following a series of design principles:

- **Few features**. The number of constructs of each language should be kept as low as possible, including only those that are important to tackle the task each language is intended to.
- **Orthogonality**. Each language should only be useful for a few tasks, avoiding redundancy with respect to the ones addressed by other languages. This will facilitate users choosing a language for each particular task.

– **Simple syntax**. The fact that few features will be included in each language will facilitate the definition of a simpler and cleaner syntax than complex languages (e.g, no need for statement separators). In addition, syntax should highlight those constructs that are the essence of each language, while "hiding" constructs that are more accidental.
– **Lightweight type information**. The amount of type information should be low. This can be achieved with type inference (which would be facilitated because languages are simple) or by relying on dynamic typing. Our current implementation uses dynamic typing, but we plan to support type inference.
– **Eclecticism**. As a major design principle, we believe that each style of model transformation has its own value to tackle certain problems, so we do not restrict Eclectic to the languages considered so far, but we are looking into other possible languages, and we are willing to contributions in this sense.

2.2 Running example

The rest of the paper will be illustrated with a transformation that takes a UML model plus an OCL model with invariants and preconditions, and generates a Java model. We have used the UML meta-model of the Eclipse UML2 plug-in, an OCL meta-model based on the ATL implementation, and the Java meta-metamodel of MoDisco. Figure 1 shows relevant excerpts of them.

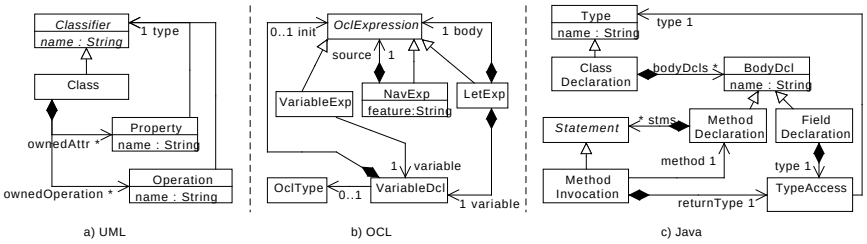

Fig. 1: Excerpts of the meta-models involved in the running example. They have been slightly simplified for the sake of clarity (e.g., renamings, hierarchy flattening).

In this transformation (uml2java) there are several aspects to consider. The mapping between UML class models and Java classes is more or less straightforward, except for some cases that requires detecting concrete patterns. On the other hand, translating OCL expressions to Java can be (partially) done with templates that generate pieces of Java code, but it requires computing type information for an accurate translation. Finally, we restrict UML models to use single inheritance.

In the rest of the section the languages are briefly introduced by showing an excerpt of the running example. The examples in this section do not consider language composition, but it is added in the next one. In addition, the development of the languages that compose Eclectic is being inspired by existing languages of

different nature. Thus, we will briefly comment on related work as the languages are explained, although more details are given in Section 2.

2.3 Mapping language

Establishing correspondences among meta-model elements in order to fix heterogeneities between semantically equivalent meta-models is the kind of transformation task that rule-based model-to-model transformation languages in the style of declarative ATL can handle naturally. For our initial prototype we have chosen to create a simplification of ATL, called SMaps.

Listing 1 shows an excerpt of the mapping between UML and Java using this language. The input and output models are indicated between parenthesis in the transformation header (line 1). This style is followed in the rest of the languages of Eclectic. As can be seen, mappings among source and target metaclasses are specified using from - to (lines 4 and 14), and they may include modifiers such as linking (stating how to relate both target elements). Mappings between structural features are specified using ←. We allow the same reference to be mapped more than once if it is multivalued (lines 10-11).

Conversions between datatypes and explicit transformations are done with the notion of converter. A converter is basically a function that is offered as a library (line 2 performs the importation) or can be specified as a mapping between datatypes (lines 20-23). It is implicitly invoked using the the convert modifier (line 7 and 8). The rationale for this notation instead of plain function calls is to enhance text clarity, so that the reader clearly identifies the left part and the right part of mapping (i.e., with a function call the right part of the mapping is "wrapped" into the actual parameters). In our implementation, libraries of converters are provided as Java classes.

```
1  mapping struct(uml) −> (java)
2    uses java_conventions
3
4    from uml!Class to cd: java!ClassDeclaration, cu : java!CompilationUnit
5      linking cd.originalCompilationUnit = cu
6
7      cd.name <− name convert java_conventions.camelCase
8      cd.visilibity <− visibility convert mapVisibility
9
10     cd.bodyDeclaration <− ownedAttribute
11     cd.bodyDeclaration <− ownedOperation
12   end
13
14   from uml!Property to m: java!MethodDeclaration, ta: java!TypeAccess
15     linking get.typeAccess = ta
16     m.name <− name convert java_conventions.getterName
17     ta.type <− type
18   end
19
20   converter mapVisibility: uml!Visibility −> String
21     #vk_public −> 'public'
22     #vk_private −> 'private'
23   end
```

Listing 1: Excerpt of the mapping from UML to Java.

The execution semantics is a mix between ATL and RubyTL. Mappings correspond to ATL rules, and the ← construct is a form of ATL binding. However, binding resolution works as in RubyTL, taking into account the conformance relationship of both the source and the target element, but, so far, only the first target element is resolved as in ATL.

With the aim reducing the cluttering of the transformation text, we have decided to do without navigation language (e.g., OCL), so that model navigation must be done in separate navigation modules.

As can be observed the language is intended neither to extract implicit information or to perform one-to-many transformations (when the number of target elements of a mapping is not known before hand). Instead, it is focused on resolving structural heterogeneities between semantically equivalent models. In this line, we expect to evolve SMaps by considering constructs in the style of the Mapping Operators (MOPs) proposed in [22].

2.4 Target-oriented language

This language, called Tao, is intended to address transformations mainly driven by the structure of the target model. This roughly corresponds to the style of model-to-text template languages, where fixed pieces of text are parameterized with expressions that fill in the holes. This kind of transformations usually has a high-degree of nesting, thus a design decision has been to consider object syntax as a way to specify large instantiation sequences (similar to QVT syntax).

Listing 2 generates a Java class from an OCL specification, with one method per OCL invariant. Templates are specified with template, and take one or more parameters, being polymorphic on the first parameter (e.g., lines 15 and 18). We have chosen this syntax because we plan to experiment with multiple-dispatch templates. The instantiation of a new object is specified with model!Metaclass { assignments }, where assignments initialize attributes and references. For example, line 6 sets the name of the Java class by combining two converters, where the result of the first one is the input of the second one. Then, line 7 initializes bodyDeclarations by creating one method per invariant defined in the OCL program. Finally, line 10 invokes mapExpression explicitly.

```
1  tao gen_java(uml, ocl) -> (java)
2    uses java_conventions, string
3
4    template mapProgram(p : ocl!Program)
5      java!ClassDeclaration {
6        name = p.name convert java_conventions.camelCase, string.concat("Check")
7        bodyDeclarations = p.invariants to java!Method {
8            name = name
9            visibility = "public"
10           expressions = p.body with mapExpression
11       }
12     }
13   end
14
15   template mapExpression(expr : ocl!NavigationExpr)
16     ...
17   end
18
```

```
19    template mapExpression(expr : ocl!LetExpr)
20      ...
21    end
```

Listing 2: Generator from OCL to Java.

As can be seen, this language has only a few elements yet it simplifies a task that is sometimes cumbersome. Please note that, although some constructs such as object-syntax or template invocation resemble parts of QVT, there is the important difference that we have avoided complex mechanisms, for instance QVT Operational initialization rules.

Finally, it could be possible to consider template languages that use the syntax of the target language (e.g., Java) to make the transformation text more fluent. In fact, these kind of languages could be integrated in Eclectic as libraries contributed by third-parties, acting as domain-specific transformation languages.

2.5 Attribute computation

Attribute grammars are a well known technique for specifying how to compute properties of language constructs, called attributes, by defining their values in terms of the attribute values of related constructs [1]. Attribute computation is defined by rules (or equations), and the attribution system is in charge of performing the evaluation by associating attribute values to syntax tree nodes, propagating values through the nodes as needed. Transformation problems that require propagating values top-down or bottom-up are typically difficult to express with some transformation languages (e.g., ATL, TGGs). In QVT it is possible to use *when* and *where* clauses to propagate values, but "propagation code" gets tangled with mapping code.

In this way, a simple language for attribute computation has been defined. It is called SAttr. It supports synthesized attributes that propagate information bottom-up, and inherited attributes that propagate top-down. It also includes a simple expression language, as the essence of this kind of transformations is to perform computations based on previously computed values. An attribution transformation is composed of attribution rules. Each attribution rule matches an element of a given metaclass and computes attribute values. To this end, there are two basic constructs: attribute initialization and attribute access. The expr[attr] ← right-part construct is used to initialize an attribute, and it has the effect of associating the value of right-part to attr for the element referred by expr. In the case of synthesized attributes, self is used to refer to the element matched by the rule. Similarly, attribute values can be accessed with the expr[attr] construct (in this case using self[attr] to access an inherited attribute). Our implementation schedules the execution according to the dependencies.

Listing 3 shows a piece of transformation that associates type information to the elements of the OCL abstract syntax model (it is partially inspired in the OCL specification). In this example, the type of each sub-expression is propagated from the leaves using the type synthesized attribute (line 2), while contextual type information is provided by the env inherited attribute (line 3, an

immutable map that associates a variable declaration with its type). In line 7, the type of a navigation expression (NavExpr) is gathered by getting the type of the receptor object (self.source[type]), and then looking up the type of the navigated feature (the feature operation is a helper defined in a navigation library). As another example, to deal with *let* expressions, line 11 adds a variable declaration to the inherited env, computing the variable type as the type of the initialization expression (so performing weak type inference). Then, it is propagated as an inherited attribute so that it is available for the body expressions. Note that, in line 16 the env attribute is used to gather the type of a variable reference that points to a variable declaration. Finally, even though it is not shown in this example, it is possible to create target elements if needed by interoperating with languages with this capacity, as is the case of Tao (see Section 4.2).

```
1  attribution typing(uml, ocl) -> ()
2    syn type : uml!Classifier
3    inh env : _!Map
4
5    rule ocl!NavExp
6      receptor[env] <- self[env]
7      self[type] <- self.source[type].feature(self.name).type
8    end
9
10   rule ocl!LetExp
11     body[env] <- self[env].put(self.varDcl, self.init[type])
12     self[type] <- self.body[type]
13   end
14
15   rule ocl!VariableExp
16     self[type] <- self[env].get(self.variable)
17   end
```

Listing 3: Collecting type information from OCL expressions.

Our current design only considers basic features of attribution systems. Other systems such as Kiama or Silver implement more complex features, and we want to explore which ones are more useful in a model transformation setting. Nevertheless, in its current state, we have found this language particularly useful to compile expression languages to a low-level representation (see Section 4.1).

2.6 Pattern matching

The languages shown so far just match a single model element. In order to address transformations where more complex patterns have to be found, we have included a simple pattern matching language in the family, named SPat.

Here we just briefly introduce the language, by means of the example shown in Listing 4, which is completed in the next section. The GettableProperty pattern gathers all public UML Property whose owning class does not contain an operation whose name would collide with the corresponding Java getter method. As can be seen this language is in the style of Tefkat, although other styles such as the one of VIATRA2 one could be possible. Interestingly, the getterName converter can be reused as a function call (line 4).

```
1  pattern GettableProperty
2    forall p: uml!Property [ p.visibility = #pk_public ] and
3    not exists o: uml!Operation [ p.owner.includes(o) and p.name = java_conventions.getterName(a.name) ]
4  end
```

Listing 4: Matching properties that do not collide with an existing "get".

In this section we have commented on four languages that illustrates the design of Eclectic. They have have been presented without taking into account how to make them work together. Next section discusses the issues involved.

3 Language composition

Our design based on a family of model transformation languages allows us to decompose a transformation problem into subproblems, where each subproblem is tackled with the most appropriate language. However, this poses two main concerns: language composition and interoperability.

In this context, *composition* is the ability of combining different languages to achieve a common task, while *interoperability* is the ability of two or more components (transformation languages in this case) to exchange information and to use it. There are two types of transformation composition: internal and external. Internal transformation composition refers to the composition of transformation constructs of a single language, while external transformation composition must take into account how to compose heterogeneous constructs belonging to different languages. Indeed, in this setting we are dealing with external transformation composition which requires interoperability.

3.1 Interoperability

Transformation language interoperability has been regarded as an important topic in model transformation [10]. However, so far, only limited forms of interoperability has been achieved [21].

Figure 2 shows the architecture of our solution. Our approach to interoperability is based on a common intermediate language, called Intermediate Dependency Code (IDC), so that each member of the family compiles down to it. IDC is composed of a few basic instructions (some of them specialized for model manipulation), which use a simplified form of Static Single Assignment (SSA) to represent data dependencies between instructions [3]. IDC does not force any particular transformation style (e.g., rule-based transformations) as it does not provide any notion of rule, but lower-level mechanisms. IDC is compiled to the JVM, and it uses a runtime library to deal with different modeling frameworks (we currently support EMF).

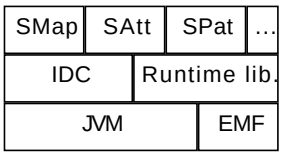

Fig. 2: Eclectic architecture

The key element of IDC is its ability to schedule the execution of several transformations based on their data-dependences using continuations. A continuation reifies the concept of "the rest of the computation", so that execution state of a given program can be saved into a continuation and restored later. This concept is supported in some programming languages, for instance Scheme or Scala [16]. Basically, we use continuations to enable a transformation execution to be suspended until a required piece of data (provided by another transformation) is available. This provides a means to integrate heterogeneous languages. Additionally, as all transformations run over the JVM, method-level interoperability is also supported. Due to space reasons we do not give a full explanation of IDC, but more details are given in [17].

3.2 Composition

Our strategy to tackle transformation composition has been to identify abstract relationships among transformation constructs, and reify them in each language as a composition mechanism. The definition of each composition mechanism has two aspects: how one language publishes or makes available the data it handles, and how another language requires and consumes this data.

Based on our experience with model transformations, and during the process of building Eclectic, we have identified four types of composition: *feeding* a transformation rule or a pattern with some value(s), *resolving* a reference from a source element to a target element, *decorating* model elements with virtual properties or operations, and *configuring* transformation definitions for execution.

Please note that this list is not exclusive, but others means of composition are possible, for instance inheritance, if one can make a transformation written in one language be extended by of another one written in another language, as proposed in [21]. In the following we discuss these forms of composition, showing how they are integrated in Eclectic.

Feeding transformation constructs. Transformation constructs such as transformation rules and patterns, are normally fed with model elements in order to start processing them. Sometimes a rule embeds the pattern (e.g., ATL), while sometimes both constructs are separated (e.g., VIATRA2). In any case, this can be seen as an abstract relationship where the transformation engine feeds some language construct with model elements (e.g., a pattern) or a transformation construct feeds another transformation construct (e.g., a pattern feeding a rule).

Both SMap and SAttr use simple patterns based on the name of a metaclass (i.e., `model!Metaclass` syntax) but if we want more complex patterns (such as the one shown in Listing 4) we need to include in each language a way to specify patterns or filter expressions. Hence, we would like a composition mechanism that does not require to change these languages to refer to a pattern expressed with SPat. Our solution has been to make the result of a pattern available as a new type, that is instantiated for each match of the pattern. Listing 5 shows the `GettableProperty` pattern, that uses the `providing` keyword to initialize the result

of the match (line 7). We have included the possibility of assigning self to some object, so that the result is referentially equal to such object (p in this case), but extended with additional properties. Now, Listing 6 uses the pattern as if it were a normal metaclass (line 5) but it is actually refering to the result of a pattern, that may be the result of performing a complex search.

```
1 spat umlext(uml)
2
3 pattern GettableProperty
4   forall p: uml!Property [p.visibility= #pk_public] and
5   not exists o: uml!Operation [ p.owner.includes(o) ...]
6
7   providing self = p,
8             self.isPublic = true
9 end
```

Listing 5: Publishing a pattern in SPat

```
smap struct(uml) -> (java)
  uses java_conventions
  uses umlext

  from umlext!GettableProperty to
        get: java!MethodDeclaration,
        ta: java!TypeAccess
    // Same as as original transformation
  end
```

Listing 6: Using a pattern in SMap

What is distinctive of this approach is that it separates patterns from rules without requiring any special syntax, but there is a seamless integration making a pattern result looks like a type. Likewise, recursive patterns in SPat are allowed using the same strategy.

Resolving references. Resolving a target element from a source element that is pointed by a reference is a primary element of model-to-model transformation languages. ATL, for instance, performs this task implicitly through a binding construct (←). In this case, we need to resolve relationships established by different languages in their own manner.

We take inspiration from Tefkat's tracking classes and the proposal of [12] for our mechanism to make source-target correspondences available. In these works, the underlying idea is to establish an interface between transformation rules by means of an intermediate model (which can be considered a trace model, where a tracking class is a type of trace link), so that there is a layer of indirection through this model to refer to the data produced by another rule. We generalize this mechanism to span several languages, making the intermediate model implicit.

In the case of SMap, the strategy is to tag each mapping, so that the set of tags of a transformation is the interface of the transformation. From an external program, a tag is as a new type of trace link that keeps a correspondence, and the interface is the set of trace link types that can be instantiated by a given transformation execution. Listing 7 shows how the SMap transformation uses a tag (classifier in line 4) to make a mapping resolvable from other transformations. Implicitly, a trace link called classifier is created, that has a source reference pointing to a UML class, and two target references pointing to the cd and cu elements (lines 5-6). Please note that in SMap a tag is not the same as a typical rule name, since the same tag may be contributed by different mappings, for instance the classifier tag applies both to mappings from UML classes to Java classes as from UML primitive types to Java.

In SAttr, however, there is no need to explicitly set the interface with tags, but it is automatically derived. In the example of Listing 3 the interface consists

of two trace links, one for each declared attribute, type and env. An external transformation would refer to an attribute value as if it were a trace link.

Transformations written in other languages, like Tao, may need to refer to these trace links. To this end, the following syntax can be used: transf!trace_link(.target)? where transf is the name of a external transformation, trace_link is the name of the trace link that will be used to resolve the source element to a target element, and target is the name of the target element to be gathered (if not given the first target element is used).

Listing 8 shows how the gen_java Tao transformation interoperates with SAttr and SMap (struct and typing transformations). Line 6 shows how to obtain the Java getter method that corresponds to a given UML property (the uml_property helper is part of a navigation library that links the OCL and UML models). This mapping between properties and methods has been performed by the struct transformation (Listing 7, lines 10-14). A more complex example is also shown in line 12, where the UML type of a *let* expression that has been computed in the typing transformation is gathered (expr[typ!type]), and from obtained type, the corresponding Java is next obtained.

```
1  mapping struct (uml) -> (java)
2  uses java_conventions
3
4  [classifier]
5  from uml!Class to cd: java!ClassDeclaration,
6                        cu : java!CompilationUnit
7    ...
8  end
9
10 [ get ]
11 from uml!Property to
12     m: java!MethodDeclaration,
13     ta: java!TypeAccess
14 end
```

Listing 7: Tagging mappings

```
tao gen_java(uml, ocl) -> (java)
uses typing, struct

template mapExpression(expr : ocl!NavExp)
  java!MethodInvocation {
    method = expr.uml_property[struct!get]
  }
end

template mapExpression(expr : ocl!LetExp)
  java!VariableDeclarationExpression {
    type = expr[typing!type][struct!classifier.cd]
  }
end
```

Listing 8: Resolving references

Decorating model elements. The possibility of adding virtual properties and operations (sometimes known as helpers) to model elements has been used so far as a way to enable navigation libraries in model transformation languages [9][15][2]. Although we have not shown any example of this scenario, Eclectic supports navigation libraries as well.

Nevertheless, we have also used this feature as a way to enable invocation of Tao templates. The interface of a Tao transformation is just the set of rules seen as operations, which return target elements. This form of interface is independent of how these operations have been implemented (with Tao in this case), but those transformations requiring explicit creation of elements simply invoke one operation expecting one or more target elemens as a result.

Listing 9 shows a piece of an SMap transformation that invokes the mapPrecondition operation to generate an assert expression from an operation precondition (lines 9-10). From the point of view of SMap, Tao templates are

converters. This means that at the SMap level the way to use languages that require explicit rule invocation is through a converter.

The SAttr language is also allowed to use Tao transformations. This is important as SAttr does not have any construct to create new elements. As we explained, SAttr provides a simple expression language, so we just rely on normal method calls as the composition mechanism.

Configuring transformation definitions The mechanisms discussed so far allow us to use the results produced by a transformation in another one written in a different language. To this end, the uses keyword establishes a dependency with an external transformation. However, we still need to configure the composite transformation which consists of the smaller transformation programs. We also would like to consider the configuration of transformation chains (i.e., feeding a transformation with the output of another).

We have devised a simple language to specify composite transformation programs and transformation chains. It basically treats transformations as functions with zero or more parameters, and with zero or more result models. There is a special construct, composite, which performs all the necessary plumbing, at the IDC level, to schedule two or more transformations to be executed together as a unit. Listing 10 shows the complete transformation chain for the UML2Java example. First of all, a new composite transformation (uml2java_si, lines 2-7), which uses the four transformations previously presented, is defined. When the same model name is used as output (lines 3-4) it means that both transformations contribute to it. If a transformation program has no output models, it is indicated using an underscore (lines 5-7). Note that we use the term transformation to refer to a piece of program that just contributes to a global result, although it does not perform any actual transformation (e.g., umlext that find patterns).

The composite transformation uml2java_si, however, does deal with multiple inheritance, so the first step in the chain would be to rewrite the UML model to remove multiple inheritance (e.g., introducing interfaces). Line 9 invokes the rewriting transformation, remove_multiple obtaining a UML model with only single inheritance (uml_sing_inh)[1]. Afterwards, the composite transformation is invoked normally (line 12), obtaining the target model. Please note that our composition mechanism is able of dealing with transformations that depend on one another. This is possible because our engine is based on continuations as explained in Section 4.1.

The task of writing transformation chains has been typically addressed with build scripts or Java programs. In our case, this language simplify writing transformation chains considering transformations as functions with a number of input/output models (in the style of MCC [11]). Besides, it provides facilities to compose transformations, and we are working on giving support to higher-order transformations.

[1] We have not implemented an in-place language in Eclectic yet, but it can be simulated with a copy transformation.

```
1  mapping struct (uml) -> (java)
2    uses gen_java
3
4    from uml!Operation to m: java!Method
5      m.name <- name
6      // The mapPrecondition template will create
7      // a java!MethodInvocation to assert the
8      // precondition (if it exists)
9      m.bodyDeclaration <- m.pre
10         convert gen_java.mapPrecondition
11   end
```

Listing 9: Invoking a template from SMap.

```
1  chain uml2java(uml, ocl) -> (java)
2    composite uml2java_si(uml, ocl) -> (java)
3      java = struct(uml)
4      java = gen_java(uml, ocl)
5      _ = typing(uml, ocl)
6      _ = umlext(uml)
7    end
8
9    uml_single_inh = remove_multiple(uml)
10   java = uml2java_si(uml_single_inh, ocl)
11 end
```

Listing 10: Configuring UML2Java.

4 Related work

The notion of family of languages has been used both to refer to independent DSLs that share common implementation artefacts [7][20] and to refer to a set of related languages that must be composed to achieve a common goal [18]. For instance, UML [14] can be considered as a family of related modeling languages, each one intended to address some concern (e.g., structure, behaviour, deployment) of object-oriented modeling, which share a common core.

In the context of model transformations there are some examples of families of languages. First of all, the QVT architecture is similar to our proposal. However, there is a significant difference, as we advocate for simple languages, while QVT Relational and Operational are complex languages. Epsilon [6] is a family of model management languages, where each language is intended for a model management task, such as validation, migration or model transformation. There is a base language, EOL, that is common to all of them. These languages, however, work independently, and they are composed by means of ANT scripts that feed one language with the ouput of another. TransML [8] is a family of languages for modeling model transformations. It is organized as a stack, with lower languages refinining the upper ones. Finally, ATL has two basic execution modes: normal mode that corresponds to model-to-model transformations, and refining mode that corresponds to in-place transformations, thus ATL can be considered as a family with two languages.

In the context of program transformation, Kiama [**?**] is a Scala library for language processing that provides several internal languages for describing attribute grammars, tree rewriting, abstract state machines, and pretty printing.

Composition of heterogeneous rule-based transformation languages is studied in [21], where a common virtual machine, called EMFTVM, is used to implement ATL and a rewriting language. Our approach is also based on an intermediate language, but with different characteristics. EMFTVM provides a common notion of module and transformation rules, which enables a common semantics for module import and rule inheritance. In our case, the intermediate language provides more general composition services (see Section 4), so that each language of Eclectic is allowed to have its own semantics. For instance, this would allow us to integrate a purely imperative language in the family (i.e., with no rules).

Creating chains of model transformations is a widely used technique to split complex transformation problems [19][11]. In these chains the input of one transformation is just fed with the output of a previous one, which has the disadvantage that the transformation execution context is lost so actual interoperability is not possible (unless complex architectures are used [23]).

5 Assessment and future work

In this paper we have presented a family of model transformation languages, called Eclectic. Our aim is to address complex transformation problems by splitting them into smaller problems that can be solved with simple languages. However, this requires a careful design of the different languages as well as taking into consideration interoperability and composability issues. Here, we have presented the design and composition mechanisms of Eclectic, which have been illustrated by means of a running example. Besides, the feasibility of the approach is demonstrated by means of an proof-of-concept implementation available at `http://sanchezcuadrado.es/projects/eclectic`.

As a summary, Table 1 relates the languages and the composition mechanisms presented in this paper. An entry with *Use* means that a language can interoperate with another language that has an entry with *Enable* for the same composition mechanism. For instance, all languages (except Chain) are able to invoke operations that decorate models elements, but Tao is the only language that currently decorates models. The SPat/Decorating entry is special because in this language only methods without side-effects are allowed, which means that it could interoperate with a navigation library but not with a Tao program.

	Feeding	**Resolving**	**Decorating**	**Composing**
Mapping (SMap)	Use	Use, Enable	Use	Enable
Attribution (SAttr)	Use	Use, Enable	Use	Enable
Target-oriented (Tao)	-	Use	Use, Enable	Enable
Pattern Maching (SPat)	Use,Enable	Use	Use*	Enable
Configuration (SChain)	-	-	-	Use, Enable

Table 1: Summary of mechanisms and languages.

As can be observed in the table, an important aspect of our design is that languages are loosely coupled, so that it is possible to evolve members of the family without affecting the other languages. In fact, new languages could be added seamlessly. This is particularly important to enable interoperability with *domain-specific transformation languages* (DSTL). We envision an scenario where part of a complex transformation is written with Eclectic, and it is completed and extended by means of a DSTL that addresses variable parts.

In principle, one possible drawback of this approach is learning facility. However, we have tried to keep the languages small and with a similar syntax so that one can learn how to use them just looking a few examples. In addition, we believe that this approach enhances intentionality of the transformation text, which favours comprehensibility.

One concern that we would like to address is the fragmentation of the transformation code. Being languages with few features, sometimes one has to rely on other languages to perform simple operations. Thus, further evaluation is needed to assess the real possibilities of the approach. One such evaluation would be to apply Eclectic to transform other UML models apart from the class diagram, testing the transformations with large models to benchmark the performance of our engine.

Finally, we have presented several proof-of-concept components of Eclectic, but further experiments are need to find out simpler and even more compact constructs. Additionally, we are looking into how to integrate other transformation styles such as in-place and bidirectional transformations.

Acknowledgements. Work partially funded by the Spanish Ministry of Economy and Competitivity (TIN2011-24139), and the R&D programme of Madrid Region (S2009/TIC-1650). This work is based on the initial results obtained during project TIN2009-11555 funded by Spanish Ministry of Economy and Competitivity. I also thank the referees for their valuable comments. Last but not least, I thank Jesús Perera Aracil for implementing the first version of the JVM compiler for Eclectic.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
2. J. Cuadrado, F. Jouault, J. Garca Molina, and J. Bezivin. Experiments with a high-level navigation language. In *Theory and Practice of Model Transformations*, volume 5563 of *LNCS*, pages 229–238. Springer Berlin / Heidelberg, 2009.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.
4. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45:621–645, July 2006.
5. Eclectic website. `http://sanchezcuadrado.es/projects/eclectic`.
6. Epsilon. `http://www.eclipse.org/gmt/epsilon`.
7. J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
8. E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos. Engineering model transformations with transML. *Software and Systems Modeling*, pages 1–23, 2011.
9. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. See also `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`. Last accessed: Nov. 2010.
10. F. Jouault and I. Kurtev. On the interoperability of model-to-model transformation languages. *Sci. Comput. Program.*, 68:114–137, October 2007.
11. A. Kleppe. Mcc: A model transformation environment. In *Model Driven Architecture - Foundations and Applications: Second European Conference, ECMDA-FA'06*, volume 4066 of *LNCS*, pages 173–187. Springer Verlag, July 2006.
12. I. Kurtev, K. van den Berg, and F. Jouault. Rule-based modularization in model transformation languages illustrated with atl. *Science of Computer Programming*, 68(3):138 – 154, 2007. Special Issue on Model Transformation.

13. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.
14. OMG. UML 2.3 specification. `http://www.omg.org/spec/UML/2.3/`.
15. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005. www.omg.org/docs/ptc/05-11-01.pdf.
16. T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proceedings of the 14th International Conference on Functional Programming*, pages 317–328, 2009.
17. J. Sanchez Cuadrado. Compiling ATL with Continuations. In *Proc. of 3rd International Workshop on Model Transformation with ATL (MtATL-2011)*, pages 10–19. CEUR-WS, 2011.
18. J. Sanchez Cuadrado and J. G. Molina. A model-based approach to families of embedded domain-specific languages. *IEEE Trans. Softw. Eng.*, 35:825–840, 2009.
19. B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers. Uniti: A unified transformation infrastructure. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.
20. M. Voelter. A family of languages for architecture description. In *8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08)*, Oct. 2008.
21. D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. Towards a general composition semantics for rule-based model transformation. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 623–637, Berlin, Heidelberg, 2011. Springer-Verlag.
22. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Surviving the heterogeneity jungle with composite mapping operators. In *Proc. of the 3rd International Conference on Model Transformation (ICMT 2010)*, pages 260–275, LNCS 6142, 2010. Springer.
23. A. Yie, R. Casallas, D. Deridder, and D. Wagelaar. Realizing model transformation chain interoperability. *Software and Systems Modeling*, pages 1–21, 2010.