RubyTL: A Practical, Extensible Transformation Language

Jesús Sánchez Cuadrado[1], Jesús García Molina[2], and Marcos Menarguez Tortosa[3]

[1] University of Murcia, Spain
jesusc@um.es
[2] University of Murcia, Spain
jmolina@um.es
http://dis.um.es/~jmolina
[3] University of Murcia, Spain
marcos@um.es

Abstract. Model transformation is a key technology of model driven development approaches. A lot of research therefore is being carried out to understand the nature of model transformations and find out desirable characteristics of transformation languages. In recent years, several transformation languages have been proposed.

We present the RubyTL transformation language which has been designed as an extensible language–a set of core features along with an extension mechanism. RubyTL provides a framework for experimenting with features of hybrid transformation languages. In addition, RubyTL has been created as a domain specific language embedded in the Ruby programming language. In this paper we show the core features of the language through a simple example and explain how the language can be extended to provide more features.

1 Introduction

The model-driven development (MDD) promotes an intensive use of models in the software life cycle. Software models are used to guide the construction of the application, and an automatic generation of source code from models is possible. At the end of 2000, OMG launched its initiative on the Model Driven Architecture^{TM} (MDA) [1], an MDD approach to address the integration challenges and the continuous changes in technology. Since then other approaches have been proposed [2][3][4], and MDD has become the new software paradigm that promises to improve software productivity and quality.

Model-to-model transformations are a key technology of the MDA approach. Most MDA research has been focused on understanding the nature of transformations and discovering desirable characteristics of model transformation languages and tools. In recent years, several transformation languages have been defined [5][6][7]. among them the QVT [8] standard proposed by the OMG. Today the success of QVT is not clear, and an alternative of a set of languages providing different styles makes sense [9].

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 158–172, 2006.

In this paper we present RubyTL, a hybrid transformation language which has been designed with three main requirements in mind: i) rapid implementation, ii) it should allow us to experiment easily with different sets of features, iii) it should provide enough functionality for writing complex transformation definitions. Three design decisions have allowed us to satisfy these requirements: the technique of embedding a domain specific language (DSL) in a programming language such as Ruby facilitates the implementation; a plugin mechanism provides a way of adding extensions, so that the language may be configured to experiment with different sets of features; finally, Ruby constructs could be used to write some kinds of complex transformations, in which a declarative style is not the most suitable. In short, RubyTL is an extensible language which provides a set of core features and an extension mechanism to add new features.

The paper is organized as follows. Section 2 describes the basic features of RubyTL transformation language, while Section 3 shows the extension mechanism. In Section 4 the transformation process is discussed. Section 5 compares RubyTL with other proposed languages. Finally, in the last section we present our conclusions and outline future work.

2 Language Description

In this section we explain the RubyTL core features, and use a transformation definition example between two simple models to illustrate the syntax and semantics of the language. These features are the basic ones for a usable transformation language, but they can be extended, as explained in Section 3.

Ruby [10] is an object-oriented programming language which is gaining constantly acceptance, especially over the last year because of the success of Ruby on Rails, a web application framework. Ruby is dynamically typed and provides an expressive power similar to Smalltalk through constructs such as code blocks and metaclasses. Because of these characteristics, Ruby is very suitable to define internal DSLs [3].

Thus, RubyTL is a model transformation language defined as a Ruby internal DSL. RubyTL is a hybrid language since it provides both declarative and imperative constructs to write transformation definitions. Like ATL [6][9], a binding construct is used to express rules in a declarative way.

The RubyTL abstract syntax, expressed as a metamodel, is shown in Figure 1. As can be seen, a transformation definition is a set of transformation rules packaged in a transformation module, and each rule has a name and four parts:

- A *from* part, where the source element metaclass is specified.
- A *to* part, where the target element metaclass (or metaclasses) is specified.
- A *filter* part, where a condition over the source element is specified, such that the rule will only be triggered if the condition is satisfied; this part is optional and if a rule has no filter it will always be triggered.

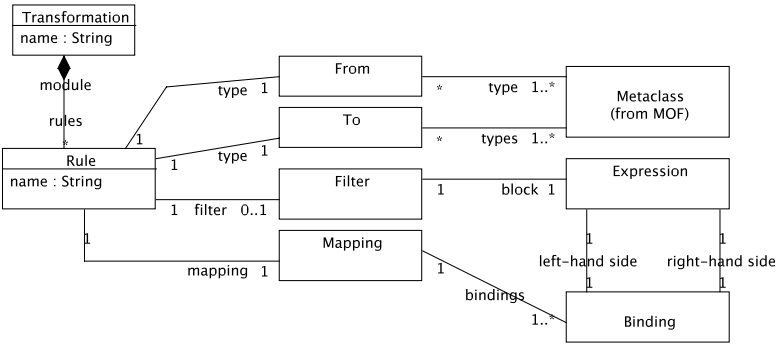

Fig. 1. Abstract syntax of RubyTL

– The *mapping* specifies relationships between source and target model elements. These relationships can be expressed either in a declarative style through of a set of *bindings* or in an imperative style using Ruby constructs. As we will explain below, a binding is a special kind of assignment that makes it possible to write *what needs to be transformed into what* instead of *how the transformation must be performed*. The declarative style is recommended, and Ruby imperative code should only be used when it is difficult to express declaratively some part of a transformation.

The concrete syntax of a RubyTL transformation definition is shown in Figure 2. It is determined by the fact that the language is implemented as a Ruby internal DSL (e.g. notice the use of `do - end` to write a code block and `| |` to set the block parameters). We have used a well-known technique to implement Ruby internal DSLs, that is, every keyword in the language is mapped to a method call and nested structures are mapped to parametrized code blocks. A discussion about the definition of Ruby internal DSLs can be found in [3].

A rule is defined by the `rule` method which expects two parameters: the rule name as a string and a code block which must have a structure conforming the concrete syntax of the rule element. The *from* and *to* parts of a rule are defined by the `from` and `to` methods, which expect as parameter a class belonging to source and target metamodels, respectively. The *filter* part of a rule is defined by the `filter` method which expects as parameter a block receiving an element of the source metaclass. The filter evaluates true if the attached block returns true, otherwise false[1]. The *mapping* part of a rule is defined by the `mapping` method which expects as parameter a block receiving the source element and one or more target elements. This block consists of either a set of *bindings* if a declarative style is adopted to implement the rule, or any other Ruby code if an imperative style is adopted. Bindings, which establish a mapping between source and target elements, have been implemented by overloading the Ruby

[1] In Ruby, the result of the last expression evaluated in a block is taken as the return value of such a block.

```
module <module-name> do

  rule <rule-name> do
    from <source-metaclass>
    to   {target-metaclass}

    filter do |source_element|
      <expression>
    end

    mapping do |<source_element>, {target_element}|
      {bindings}
      # bindings has the form:
      #     target_element.property = source_element.property
    end
  end

  # one or more rules
end
```

Fig. 2. Concrete syntax of RubyTL. In this notation `<>` means one ocurrence and {} means one or more ocurrences.

assignment operator. It is worth noting that RubyTL is easy to learn and, since a new notation has been built on top of Ruby, only a little knowledge of the Ruby language is required.

2.1 Example

Once we have outlined the structure of the language, we show an example of transformation definition and explain some language features. The example is a simple transformation from a class model to a Java model, such that i) each class is transformed to a Java class, ii) each public attribute of a class is transformed to a pair of get/set methods plus a private field in the Java class, and iii) each private attribute of a class is transformed to a private field in the Java class.

Figure 3 shows the source (Class) and target (Java) metamodels [11]. Class metamodel is defined inside a package named `SimpleClass`. According to this metamodel, a class is composed of attributes; an attribute has a name and a visibility and the type of an attribute can be a class or a primitive type. Java metamodel is defined inside a package named `SimpleJava`. According to this metamodel a Java class is composed of features which can be fields or methods; a method can have zero or more parameters; both features and parameters are typed, therefore they inherit from `TypedElement`, which gives them a type and a name.

The following transformation definition expresses the transformation from class model to Java model, as explained above. In *http://gts.inf.um.es/downloads*

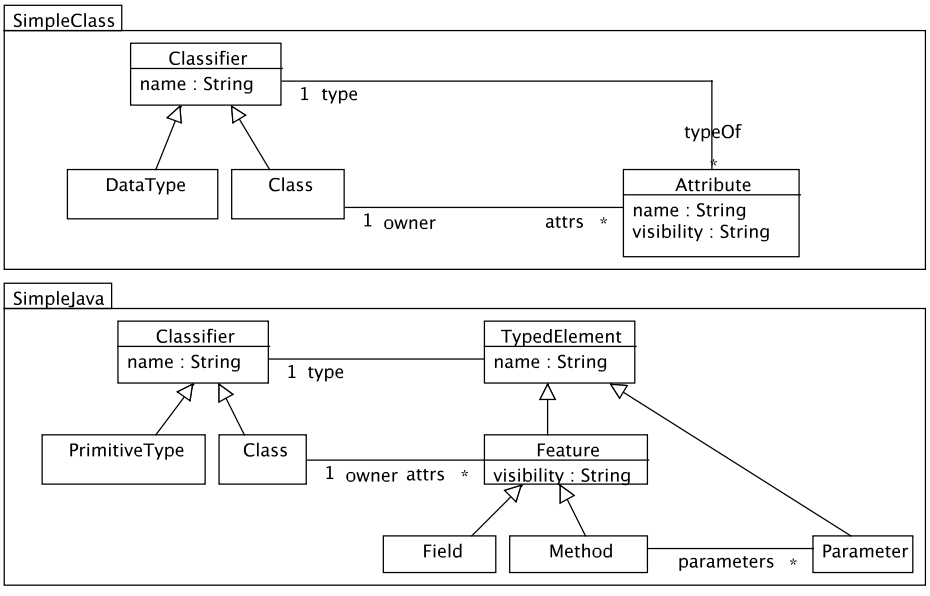

Fig. 3. Class metamodel and Java metamodel

a more complex version of this transfomation example, in which operations are introduced in the source metamodel, can be found.

```
module Transformation

    rule 'klass2javaclass' do
        from    SimpleClass::Class
        to      SimpleJava::Class
        mapping do |klass, javaclass|
            javaclass.name = klass.name
            javaclass.features = klass.attrs
        end
    end

    rule 'attribute2features' do
    from SimpleClass::Attribute
    to   SimpleJava::Field, SimpleJava::Method, SimpleJava::Method
        filter  do |attr|
            attr.visibility == 'public'
        end
        mapping do |attr, field, get, set|
            field.name = attr.name
            field.type = attr.type
```

```
            field.visibility = 'private'
            get.name = 'get'  + attr.name
            get.type = attr.type
            get.visibility = 'public'
            set.name = 'set' + attr.name
            set.visibility = 'public'
            set.parameters = attr.type
        end
    end

    rule 'attribute2field' do
        from    SimpleClass::Attribute
        to      SimpleJava::Field
        filter  do |attr|
            attr.visibility == 'private'
        end
        mapping do |attr, field|
            field.name = attr.name
            field.type = attr.type
            field.visibility = 'private'
        end
    end

    rule 'type2parameter' do
        from    SimpleClass::Classifier
        to      SimpleJava::Parameter
        mapping do |classifier, parameter|
            parameter.name = 'value'
            parameter.type = classifier
        end
    end

    rule 'datatype2primitive' do
        from    SimpleClass::DataType
        to      SimpleJava::PrimitiveType
        mapping do |src, target|
            target.name = src.name
        end
    end
end
```

A key point of the example is the binding construct. For instance, the binding `javaclass.features = klass.attrs` establishes a mapping from class attributes to Java features and yields to the execution of a rule that specifies such mapping. In this case both `attribute2features` and `attribute2field` rules are valid choices, but the filter of these rules allows the selection of only one,

depending on the attribute visibility. If there are more than one possible choice, the decision of which rule will be selected depend on which plugins are installed. The default plugin simply raises an error if this occur, but a more complex plugin could provide the developer a mechanism to resolve such situation.

Note that bindings established between primitive types (e.g. `field.name = attr.name`) do not involve any rule invocation since they belong to the same underlying meta-metamodel.

It is worth mentioning how clear and legible the transformation shown above is. The non-intrusive Ruby syntax and the combination of code blocks and methods have allowed us to design a very clean language. An important feature of RubyTL, which makes it a clean language, is the implicit rule application driven by the bindings established between model elements. The order in which the rules are written in the transformation definition is irrelevant. Below, we discuss some features of the language, and use the example to explain them.

2.2 Naming Metaclasses

In the rules of the example, notice how the metamodel classes (metaclasses) are named in the *from* and *to* parts: the name of the metaclass is prefixed by the name of the package in which that metaclass is enclosed plus two colons. This usual notation can be used because the metaclasses organization in packages is replicated in Ruby as classes enclosed in modules (the name of a class is prefixed by the name of its module). For example, in RubyTL the `Attribute` metaclass enclosed in the `SimpleClass` package can be named as `SimpleClass::Attribute` because of a class named `Attribute` has been created within a module named `SimpleClass`.

2.3 Expressions

Ruby expressions are used to write filters and bindings. For example, in the `attribute2features` rule a simple example of filter expression can be seen: `attr.visibility == 'public'` checks if the attribute visibility is public.

It is very usual among transformation languages to use OCL as a query language to navigate source metamodels and to express conditions. RubyTL does not use any OCL-like query language since Ruby provides a powerful library for managing collections. This library offers great expressive power for writing expressions, due mainly to the existence of internal iterators. For example, `klass.attrs.select {|attr| attr.visibility == 'public'}` collects all the public attributes of a class.

2.4 Bindings and Rule Conformance

As we have noted, the mapping of a rule is composed of a set of bindings. The purpose of a binding is to specify a relationship between source and target elements and it is written as an assignment in the form `target.property = source-expression`, where:

- **source-expression** is a Ruby expression whose result is an element, or a collection of elements, belonging to the source model. Therefore, the type of the right-hand side of the assignment is given by the type (metaclass) of source-expression.
- **target** is a parameter of the mapping code block; this parameter denotes a target element to be created and its type is given in the *to* part of the rule.
- **property** must be a property of the previously created target element. The type of the left-hand side of the assignment is given by the type of the metamodel feature to which the property corresponds.

The definition of binding semantics is based on the "conforming rule" concept: "A rule conforms to a binding if the type in its *from* part conforms to the type in the right part of the binding assignment and the type in its *to* part conforms to the type in the left part of the binding assignment". The semantics of a binding can be defined as "there exists a conforming rule which transforms the type of the right-hand side of the binding assignment into the type of the left-hand side of the binding assignment".

In the example, the binding `javaclass.features = klass.attrs` in the `klass2javaclass` rule means that there exists a rule whose *from* part conforms to `SimpleClass::Attribute` and its *to* part conforms to `SimpleJava::Feature`. It is important to note that conformance between types must take into consideration inheritance between metaclasses, that is, a subtype conforms to its parent type. For example, the `attribute2features` rule conforms to the previous binding: its *from* part is `SimpleClass::Attribute` and its *to* part conforms to `SimpleJava::Feature` as both `SimpleJava::Field` and `SimpleJava::Method` are subtypes of `SimpleJava::Feature`.

A transformation definition is well-formed if for each binding involving two non-primitive types, as left-hand and right-hand side types, there exist one or more conforming rules but there is one and only one applicable rule. This means that if two or more conforming rules exist, their filter conditions must be exclusive, since only one of them can be applied. Since RubyTL is an embedded DSL, checking if a transformation definition is well-formed must be done at runtime.

2.5 Rule Evaluation

The evaluation of a transformation definition is driven by the bindings established between source and target elements. Assignment operator has been overloaded in such a way as to look for the correct rule to transform the right part of the binding assignment into the left part. Whenever a conforming rule is found it is applied using the element in the right part of the binding as the source element. If the type of the right-hand side element is a collection then it will be flattened and the rule will be applied once for every single element.

Every transformation must have an entry point in order to start the evaluation. The entry point is the first rule which is applied to all existing elements of the metamodel class specified in its *from* part (in the example it is applied to all instances of `SimpleClass::Class`). In Section 3 the language is modified to allow different entry points.

Applying a rule is simply executing the code block of its mapping part. Just before a rule is applied, new target elements are created–one element for each metaclass specified in the *to* part of the rule. While the first parameter of the mapping code block receives the source element, the rest of parameters receive the target elements created as a result of the rule execution. In the example, the mapping code block of the rule `attribute2features` has four parameters: `attribute` whose type is `SimpleClass::Attribute`, `field` whose type is `SimpleJava::Field`, and `get` and `set` whose type is `SimpleJava::Method`. We refer to the first parameter as *source parameter* and the rest as *target parameters*.

Execution of a rule returns one or more target elements which are assigned to the target feature related to the binding which triggers the rule. An important consideration is that a source element is never transformed twice by the same rule, that is, if a source element has been already transformed by a rule the previous result is returned. In the example, when the `attribute2features` rule is applied, the result of the binding `field.type = attribute.type` is stored and it is returned as the result of `get.type = attribute.type` when that rule is applied to resolve such binding.

This way of evaluating rules is applied when the rules are written in a declarative style based on bindings. Since the evaluation algorithm simply executes the Ruby code written in the mapping code blocks (notice bindings are a Ruby construct), it is also possible to write any Ruby construct inside the mapping part of a rule, thus yielding an imperative style.

2.6 Reflection

Another property of RubyTL is that it can be used in a reflective way. Just like reflective languages such as Java or Ruby, the main concepts of the language (transformation, rule, mapping and metaclass in this case), except binding, can be manipulated in runtime since they are Ruby objects. Therefore, they can be handled by RubyTL rules, making it possible to write a RubyTL transformation that takes another RubyTL transformation as input and generates a modified RubyTL transformation as output.The main limitation is that reflectivity cannot deal with bindings, since they are actually Ruby code. This makes that the output transformation cannot be serialized, but only used in runtime.

To sum up, RubyTL is an unidirectional hybrid language, which relies on the concepts of rule and binding to specify a transformation. Rules are resolved implicitily and in a deterministic way. Figure 4 shows the core features of RubyTL through a feature diagram according to [12].

3 Extension Mechanism

RubyTL is an extensible language, that is, the language has been designed as a set of core features with an extension mechanism. In the previous section we have explained the core features, and in this section we will present the extension mechanism based on the use of plugins.

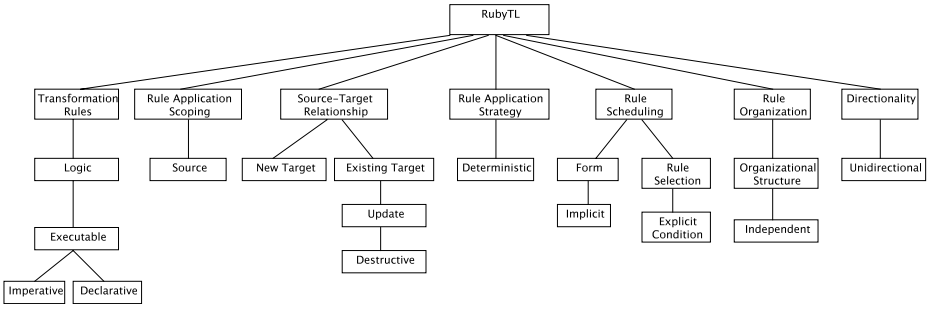

Fig. 4. Feature diagram showing the core features of RubyTL according to [12]

A plugin is a piece of Ruby code which modifies the runtime behaviour of the language by acting on the language syntax, the evaluation engine or even the model repository. The language can be considered a framework with a set of extension points that plugins can implement to add functionality. Some examples of additional features are the following: definition of new kinds of rules with a different behaviour, adding or removing syntax elements, renaming existing keywords, and modifying the transformation algorithm. Adding a new language feature is as simple as creating a plugin which implements a few extensions points. Obviously, a new feature can only be added if the necessary extension points have been planned.

The underlying idea behind this plugin mechanism is to have an extensible language intended to experiment with transformation languages features. Given a transformation problem, different combinations of features could be tried out in order to decide which is the most appropiate. Before the evaluation of a transformation, the developer should select the set of suitable plugins so that the language is properly configured. Each time a set of plugings is installed it is as if a new instance of the language were created.

Next we outline some advanced features implemented as plugins. In addition to implicit rule execution (expressed through bindings), it is possible to call rules explicitly by their name. A plugin traverses all rules in the transformation and creates a method with the same name of the rule which can be explicitly invoked. This plugin allows to call rules when mappings are written in an imperative style.

As mentioned before, the entry point of a transformation is the first rule. This behaviour is generalized by a plugin which implements a new kind of rules, named *top rules*. A top rule is always applied to all instances of the type specified in its *from* part, thus a transformation definition could have more than one entry point.

A rule never transforms a source element twice, and this is the behaviour that is usually expected. However, it may be necessary for a rule to be evaluated more than once for a particular source element (in ATL this is the default behaviour of rules). In order to provide this behaviour, we have implemented a plugin by adding a new kind of rule, named *creator*.

Another plugin allows mappings not to be restricted to one-to-one mappings, but it is possible to perform one-to-many and many-to-one mappings. At this moment, we are exploring different ways of writing such mappings in a declarative and readable manner. Finally, adding traceability support has been quite easy with a plugin.

In [12] several variation points in transformation languages are identified. Some of these variation points could be extensions to RubyTL, and they are summarized in Figure 5. For instance, the language can be modified to perform a transformation in several phases, where each phase has a specific purpose and only certain rules can be invoked in a given phase. It would allow us to think about a transformation as a set of refinement steps o phases, where each phase rely on the job accomplished by previous phases to complete its job.

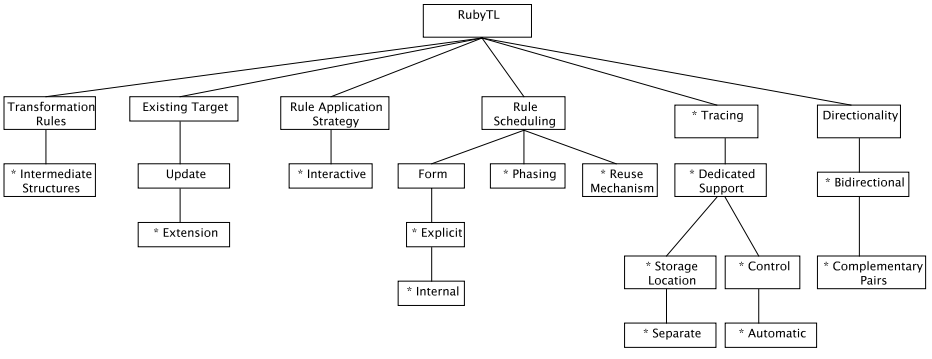

Fig. 5. Feature diagram showing possible language extensions according to [12]. Features marked * are suitable to be implemented as plugins.

There are several advantages of this extensible language approach. First, we have an environment in which to experiment mixing transformation languages features and where new features can be implemented if required. Second, implementation and maintenance are easier due to the modular design. Finally, both experimenting with features and even implementing new features does not require any knowledge about language internals. In addition, the fact that RubyTL is an internal DSL makes the plugin mechanism easy (e.g. modifying the language syntax in runtime).

4 Transformation Process

RubyTL has been implemented as a Ruby internal DSL. This key design choice means we are relying on the Ruby interpreter to parse and evaluate the transformation definition. The transformation engine and the XMI parser has also been implemented in Ruby.

Figure 6 is a process diagram which shows the components and the data involved in the whole transformation process. The steps are the following:

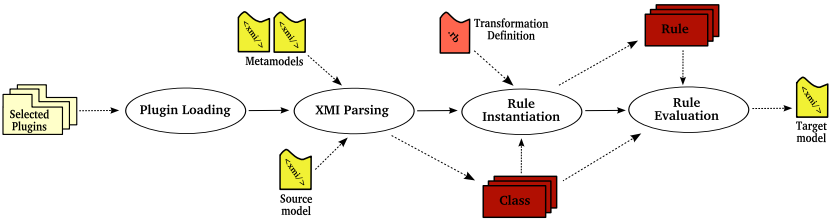

Fig. 6. Execution of RubyTL transformation engine

1. Since RubyTL has a pluggable design, the first step is to load the suitable plugins to configure the language with certain features. The user should select the plugins to be loaded, and the plugin mechanism check dependencies between them.
2. Source metamodel, target metamodel and source model are *xmi* files. A parser written in Ruby reads these input files and a set of Ruby classes are generated and loaded in the Ruby interpreter. These classes correspond to the classes defined in the source and target metamodels.
3. Once metamodels have been loaded, the transformation definition (it is in effect Ruby code) is read by the Ruby interpreter itself, which leads to the creation of a set of rule objects. These rules will be used by the transformation engine to perform the transformation.
4. As explained above, the transformation execution is driven by the bindings established in the mapping part of the rules. As the rule evaluation is being performed plugins implementing extension points could be called. For instance, if a plugin implements a strategy to choose between two or more applicable rules, it will be called when more than one rule can be applied.
5. The output of the transformation process is an *xmi* file containing a target model conforming to the target metamodel.

5 Related Work

Several classifications of model transformation approaches have been developed [12][13][14]. According to these classifications, the different model-model approaches can be grouped into three major categories: imperative, declarative and hybrid approaches. Imperative approaches are focused on *how* the transformation is done; the direct model manipulation approach is the most common mechanism which uses programming languages such as Java and procedural APIs. Declarative approaches, such as relational, functional or graph-based approaches, are focused on *what* the transformation does. Finally, hybrid approaches combine declarative and imperative constructs.

Some of the latest research efforts in model transformation languages are ATL, Tefkat, MTL and Kermeta. MTL and Kermeta [5][15] are imperative executable metalanguages not specifically intended to model-model transformation,

but they are used because the versatility of their constructs provides great expressive power. However, the verbosity and the imperative style of these languages make writing complex transformations difficult because they are very large and not readable.

ATL is a hybrid language with a very clear syntax [6][9]. It includes several kinds of rules that facilitate writing transformations in a declarative style. However, the complete implementation of the language is not finished yet, and at the moment only one kind of rule can be used. Therefore it may be difficult to write some transformations declaratively. ATL and RubyTL share the same main abstractions, i.e. rule and binding, but ATL is statically typed while RubyTL uses dynamic typing. Static typing allows ATL to perform compile time checks, for instance to do optimizations. On the other hand, dynamic typing is less restrictive and offers more flexibility, which is very important for an extensible language such as RubyTL.

Tefkat is a very expressive relational language which is completely usable [7]. As noted in [16], writing complex transformations in a fully declarative style is not straightforward, and the imperative style may be more appropriate. That is why supporting a hybrid approach is a desirable characteristic for a transformation language, to help in writing practical transformation definitions. Tefkat only supports the declarative style, which could be an important limitation.

In [16], a set of quality requirements for a transformation language is presented. If RubyTL is evaluated against these requirements the following are found. Usability is facilitated providing a clear syntax and a style of writing transformation definitions appropriate to the usual background of the developers. Furthermore there is a good trade-off between conciseness and verbosity because it is a hybrid language–a declarative style allows rules to be written in a concise way and a more verbose imperative style can be used when is needed. Regarding scalability, the use of a native EMOF[2] repository provides a good performance, and it can cope with large transformations without loss of performance due to the nature of the language itself.

6 Conclusions and Future Work

In June 2005 we started a project for the creation of a framework intended to experiment with ATL-like transformation languages features, that is, features of hybrid languages in which the declarative style is expressed by a binding construct. The result of this project have been RubyTL, an extensible transformation language. We have gone through the following steps.

1. We observed that the technique of embedding a DSL in a programming language such as Ruby provided three important advantanges: i) a fast implementation, ii) changes in the language could be easily made, and iii) Ruby constructs can be used to write complex transformations.

[2] http://rmof.rubyforge.org/

2. We also realized that Ruby facilitates the creation of a plugin mechanism, so that RubyTL could be designed as an extensible language. We established the core features and the extension points, and we implement a set of plugins.
3. Finally, we have experimented with the language by writing transformation definitions.

In this paper we have presented RubyTL core features and the plugin mechanism. We have used a classical example for describing the language. This example has illustrated that RubyTL transformation definitions are readable and easy to understand because of the declarative style of the language. An imperative style could be adopted for complex transformations by writing Ruby code. Therefore, RubyTL is a fully usable language to write transformations of any level of complexity. But the main novelty of RubyTL is to provide a framework in which to experiment with features of hybrid transformation languages and to extend the language without taking into account its internals.

The fact that RubyTL is implemented as a Ruby internal DSL causes some limitations. The main drawback is that there is not a static type checking, due to Ruby being a dynamically typed language and it may make a good tool support difficult. In any case, we are currently working on the integration of our transformation engine inside the Eclipse platform by using RDT[3]. At this moment, an editor with syntax highlighting, a launcher for transformation definitions, and a configuration tool for plugins is available[4]. As future work, we expect to be able to provide a debugger to RubyTL, and we are exploring the possibility of using RubyTL to refactor Ruby code.

We will continue writing transformation definitions in the context of real applications to find problems which require new constructs in order to be declaratively specified.

Acknowledgments

This work has been partially supported by Fundación Seneca (Murcia, Spain), grant 00648/PI/04, and Consejera de Educación y Cultura (CARM, Spain), grant 2I05SU0018. Jesús Sánchez enjoys a doctoral grant from the Spanish Ministry of Education and Science.

References

1. Object Management Group. MDA Guide version 1.0.1. omg/2003-06-01, 2003. OMG document.
2. Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[3] http://rubyeclipse.sourceforge.net/
[4] http://gts.inf.um.es/downloads

3. Martin Fowler. Language workbenches: The killer-app for domain specific languages?, June 2005. http://www.martinfowler.com/articles/languageWorkbench.html.
4. Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodelling, A Foundation for Language Driven Development*. Xactium, 2004.
5. Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, 2005.
6. Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *OOPSLA 2003 Workshop*, Anaheim, California, 2003.
7. Michael Lawley and Jim Steel. Practical declarative model transformation with Tefkat. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, 2005.
8. OMG. Revised submission for MOF 2.0 Query/View/Transformation, 2005. http://www.omg.org/cgi-bin/apps/doc?ad/2005-03-02.
9. Frédric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.
10. D. Thomas. *Programming Ruby. The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.
11. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
12. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Technique in the Context of the Model Driven Architecture*, Anaheim, October 2003.
13. Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. Review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards final standard, 2003.
14. Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September/October 2003.
15. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.
16. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In *International Workshop on Graph and Model Transformation (GraMoT). A satellite event of the Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September 2005.