# A plugin-based language to experiment with model transformation

Jesús Sánchez Cuadrado and Jesús García Molina

University of Murcia, Spain,
{jesusc, jmolina}@um.es

**Abstract.** Model transformation is a key technology of model driven software development approaches. Several transformation languages have appeared in the last few years, but more research is still needed for an in-depth understanding of the nature of model transformations and to discover desirable features of transformation languages. Research interest is primarily focused on experimentation with languages by writing transformations for real problems.
RubyTL is a hybrid transformation language defined as a Ruby internal domain specific language, and is designed as an extensible language: a plugin mechanism allows new features to be added to core features. In this paper, we describe this plugin mechanism, devised to facilitate the experimentation with possible features of RubyTL. Through an example, we show how to add a new language feature, specifically we will develop a plugin to organize a transformation in several phases. Finally, we discuss the advantages of this extensible language design.



## 1 Introduction

Model transformation is a key technology for model driven software development (MDD) approaches to succeed. As a result of academic and industrial efforts, several model transformation tools and languages have appeared in the last few years, but more research is still needed for an in-depth understanding of the nature of model transformations and to discover the essential features of transformation languages. Therefore, the research interest of the MDD area is focused on experimentation with existing languages, by writing transformation definitions for real problems. Theoretical frameworks such as the feature model discussed in [1] and the taxonomy of model transformations presented in [2] are very useful to compare and evaluate design choices during experimentation.

A year ago, we started a project for the creation of a tool to experiment with features of hybrid transformation languages whose declarative style is supported by a binding construct, as is the case of the ATL language [3]. The result of this project is RubyTL [4], an extensible transformation language created by the technique of embedding a domain specific language (DSL) in a dynamic programming language such as Ruby [5]. RubyTL supports extensibility through a plugin mechanism: a set of core features can be extended with new features by creating plugins which implement predefined extension points.

In this paper, we present the extensible design of RubyTL, analyzing the extension points identified from the transformation algorithm. In addition, we show how to add a plugin for the needs of a particular problem, concretely a plugin that organizes a transformation in several phases; the extension points involved in this plugin are identified from its requirements.

The paper is organized as follows. The next section presents the core features of the RubyTL language and the transformation algorithm. Section 3 describes how the plugin mechanism is implemented, while Section 4 describes the plugin example. In Section 5 related work is discussed. Finally, conclusions and future work are presented in Section 6.

## 2 Language and Algorithm

RubyTL is a model transformation language designed to satisfy three main requirements: i) according to the recommendations exposed in [6][7], it should be a hybrid language, because declarative expressiveness may not be appropriate for complex transformation definitions, which may require an imperative style; the declarative style is provided by a binding construct similar to that in the ATL language [3], ii) it should allow easy experimentation with different features of the language, and iii) a rapid implementation should be possible . These requirements have been satisfied through two key design choices: the definition of the language as a Ruby internal DSL and the implementation of a plug-in extension mechanism. RubyTL provides a set of core features, and new features can be added by plugins connected to a set of predefined extension points. In this section we describe the core features and the transformation algorithm, whereas the plugin mechanism will be explained in the following section.

As said in [8] the internal DSL style is much more achievable in dynamic languages like Lisp, Smalltalk or Ruby. We have chosen Ruby, but the approach is language independent. Ruby is a dynamically typed language which provides an expressive power similar to Smalltalk through constructs such as code blocks and metaclasses. Because of these characteristics, Ruby is very suitable for embedding DSLs [8], so that Ruby internal DSLs are being defined in areas such as project automation and electronic engineering [9]. Applying this technique to create RubyTL has allowed us to have a usable language in a short development time. Moreover, Ruby code can be integrated in the DSL constructs, so that the hybrid nature can be obtained in a uniform way: everything is Ruby code.

Below we show a simple example of a transformation definition to illustrate the core features of RubyTL, and then explain the basis to understand the language. A more detailed explanation about the core language can be found in [4]. We have considered a classical transformation problem: the class-to-table transformation, whose metamodels are shown in Figure 1.

```
rule 'klass2table' do
  from  ClassM::Class
  to    TableM::Table
```

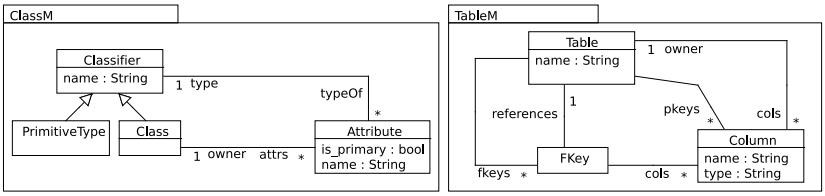


**Fig. 1.** Class and relational metamodels used in the example.

```
  mapping do |klass, table|
    table.name = klass.name
    table.cols = klass.attrs
  end
end

rule 'property2column' do
  from    ClassM::Attribute
  to      TableM::Column
  filter  { |attr| attr.type.kind_of? ClassM::PrimitiveType }
  mapping do |attr, column|
    column.name = attr.name
    column.type = attr.type.name
    column.owner.pkeys << column if attr.is_primary
  end
end

rule 'reference2column' do
  from    ClassM::Attribute
  to      Set(TableM::Column)
  filter  { |attr| attr.type.kind_of? ClassM::Class }
  mapping do |attr, set|
    table = klass2table(attr.type)
    set.values = table.pkeys.map do |col|
      TableM::Column.new(:name => table.name + '_' + col.name,
                         :type => col.type)
    end
    table.fkeys = TableM::FKey.new(:cols => set)
  end
end
```



As can be seen in the example, a transformation definition consists of a set of transformation rules. Each rule has a name and four parts: i) the *from* part, where the source element metaclass is specified, ii) the *to* part, where the target element metaclass (or metaclasses) is specified, iii) the *filter* part, where the condition to be satisfied for the source element is specified, and iv) the *mapping* part, where the relationship between source and target model elements are expressed, either in a declarative style through of a set of bindings or in an

imperative style using Ruby constructs. 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. A binding has the following form `target_element.property = source_element`, for instance in the `klass2table` rule.

In the example, the first rule (`klass2table`) will be executed once for each element of type `Class`, leading to the creation of an element with type `Table`. In the *mapping* part of this rule, relationships between class features and table features are specified. In particular, it is important to note how the `table.cols = klass.attrs` binding will trigger the execution of the `property2column` rule or `reference2column` rule for it to be resolved.

The `property2column` and `reference2column` rules illustrate how imperative code can be written within a mapping part. In the `property2column` rule, two bindings are followed by a Ruby sentence which checks if an attribute has to be converted into a primary key to add the column to the set of table primary keys. The `reference2column` rule is an example of a one-to-many relationship (one-attribute to many-columns). In this case, all the mapping is written in an imperative way, as the set of columns which take part in the foreign key is explicitly filled.

This example raises an important question about querying the target model. Using the expression `table.pkeys.map`, in the `reference2column` rule, we are relying on the target model to calculate the foreign key columns, which could cause problems because we are navigating on a partially generated model. In this case, if circularity exists in the source model, it may cause the primary keys of a table to be partially calculated when they are used to generate foreign key columns. Of course, this simple example can be solved without relying on the target model but more complex transformations, like the one proposed in [10], could become difficult.[1]

In Section 4 we propose a language extension to address the problem of navigating the target model. With this extension, a transformation would be able to safely navigate the target model.

### 2.1 Transformation algorithm

The execution model of RubyTL can be explained through a recursive algorithm. As we will see in the next section, this algorithm is the basis to identify extension points.

Every transformation must have at least one entry point rule in order to start the execution. By default, the first rule of a transformation definition is the entry point rule, for instance the `klass2table` rule in the example. When the transformation starts, each entry point rule is executed, by applying the rule to all existing elements of the metamodel class specified in its *from* part (in the example, to all instances of `ClassM::Class`).

[1] In http://gts.inf.um.es/downloads examples and a more detailed explanation about this issue can be found.

The structure of the main procedure of the transformation algorithm is a loop executing the set of entry point rules. For each entry point rule, target model elements are created for each source model element satisfying the rule filter, and then the rule is applied, i.e. the mapping part is executed.

```
Transformation entry point()
  entry-rules = select entry point rules
  for each rule R in entry-rules
      source-instances = get all instances of source type of rule R
      for each instance S in source-instances
          if S satisfy the rule R filter
              T = create target instances
              apply rule(R, S, T)
```

The `Apply rule` iterates over each binding in the mapping part of a rule, distinguishing two cases: a primitive value must be assigned to the target element property or the binding must be resolved by applying other rules. As said, a binding has the following form: `T.property = S`, therefore `S` and `T` are part of a binding.

```
Apply rule(R : rule to apply,
           S : source element, T : target elements)
  for each binding B in R.bindings
      if B is primitive then assign value to property
                        else resolve binding(B)
```

The `Resolve binding` procedure below shows how the binding resolution mechanism acts in two steps. Firstly, all the rules conforming the binding and satisfying the rule filter are collected. Secondly, for each selected rule, proper target elements are created and linked, and then the rule is applied.

```
Resolve binding(B : binding)
   S = source element of B
   T = target element of B
   P = property of T taken from B
   C = list of conforming rules initially empty

   for each rule R in the set of transformation rules
       if R is conforming with B and R satisfy filter
          add R to C

   for each rule R in C
       T' = create target instances
       link T' with P of T
       apply rule(R, S, T')
```

It is worth noting the recursive nature of the algorithm and how such recursion is implicitly performed by means of bindings. The recursion finishes when a mapping is only composed of primitive value assignments. Another way to finish

recursion could be by preventing a rule from transforming the same source element twice. This key feature, which allows the language to deal with metamodels having cycles, has been added by a plugin.

## 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 this section we will present the extension mechanism based on plugins. First, we will explain the underlying ideas behind the design of our extensible language, then we illustrate the extension points we have identified.

From the transformation algorithm shown in the previous section, we have identified some parts in the transformation process which are variable, and depending on how they are implemented, the transformation algorithm will behave differently. These variable parts will be extension points. Since the transformation algorithm is general, it can be implemented in any general purpose programming language. What would change from one implementation to another would be: (a) the way extension points are defined and implemented, and (b) the concrete syntax of the language.

A plugin is a piece of code which modifies the runtime behaviour of RubyTL by acting either on the language syntax, the evaluation engine or even the model repository. The language can be considered as a framework providing a set of extension points that plugins can implement to add functionality. According to the language aspect being extended, there are three categories of extension points: (1) related to the algorithm, (2) related to creation of new rules and management of the rule execution cycle, and (3) related to the language syntax. These categories are explained in detail in the following subsections.

Regarding how those extension points are implemented, there are two kinds of extension points: *hooks* and *filters*. *Hooks* are methods which can be overridden to implement a new functionality (similar to the *template method* design pattern [11]), while *filters* follow the same schema as web application filters [12]. Filters allow plugins to collaborate in a certain extension point. In addition, a filter can be seen as an application of the *Observer* pattern [11], as it allows a plugin to register for events occurred in the transformation process.

An extension point always has a corresponding hook, and can also have two filters: a filter which is called just *before* the hook extension point is invoked, and a filter which is called just *after* the hook extension point has been invoked. The reason to use hooks and filters is because if two plugins implement the same hook they could be incompatible, but with filters two or more plugins could share the same extension point. Sometimes, certain extensions can be implemented by a filter without overriding existing extensions.

We will call hook based extension points *hook extension points* and filter based extension points *filter extension points*.

### 3.1 Algorithm extensions

Extensions related to the transformation algorithm are directly based on the three procedures explained in Section 2. Each of these procedures is a hook extension point itself. Moreover, some parts of these procedures are also hook extension points, as can be seen in Figure 2, where the execution order of the extension points is shown.

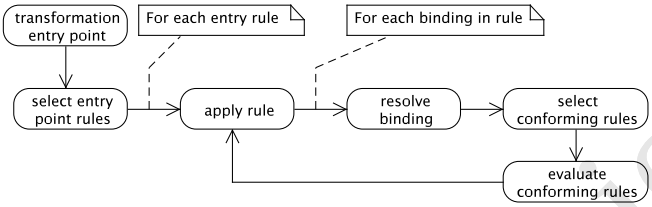


**Fig. 2.** Execution order of algorithm extension points

Therefore, the available extension points are the following:

- **transformation entry point**. This extension point corresponds to the `Transformation entry point` procedure. It provides a way to apply entry rules in a different manner, as will be shown in the example in Section 4. This extension point has a nested hook extension point, *select entry point rules*.
- **select entry point rules**. The application of entry point rules depends on how they are selected. This extension point allows us to apply different selection strategies, for instance based on a special kind of rule. Usually, it is necessary to declare more than one entry point rule, which can be provided by a plugin implementing this extension point.
- **apply rule**. This extension point, which corresponds to the `Apply rule` procedure, specifies how a rule should be applied (by default executing its mapping part). Since this behaviour could depend on the kind of rule being applied, a similar extension point is included in the rule extension point category, so that the default behaviour of this extension point is delegating to this rule extension point.
- **resolve bindings**. This extension point, which corresponds to the `Resolve binding` procedure, specifies how a binding is resolved by selecting and evaluating conforming rules, so that it delegates in two nested hook extension points: *select conforming rules* and *evaluate conforming rules*.
- **select conforming rules**. It is intended to specify how to select rules conforming a binding. It is useful to change the conformance strategy, as will be shown in the example in Section 4.
- **evaluate conforming rules**. It is intended to specify how conforming rules are evaluated. This evaluation may require creating new target elements. In

the plugin example of the next section, this extension point must be implemented because phasing needs an evaluation procedure which is different from the default.

Regarding filter extension points, two filters not related to hook extension points are defined, at the beginning and at the end of the transformation. The first one allows us to set up global information before the transformation starts, for instance, a new model could be created in the model repository to store the transformation trace model. The second one allows us to perform "cleaning" activities after the transformation has finished. In addition, the following filters related to the identified hook extension points are defined: after select entry rules, before/after apply rule, before/after select available rules, before/after evaluate available rules.

### 3.2 Rule extensions

Rule extensions points are intended to create new kind of rules, which will have a different behaviour than the default one. These extension points are related to the rule execution cycle (i.e. the set of the states a rule passes through). According to the transformation algorithm, a rule passes through the states shown in Figure 3. Each one of these states (except *waiting to be executed*) is an extension point itself. Actually, these steps are driven by the algorithm (see Figure 2) which is in charge of delegating to the proper rule extension point in each step of its execution.

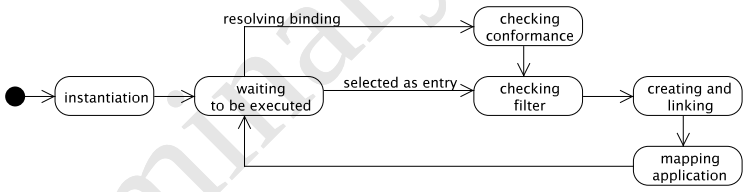


**Fig. 3.** Rule execution shown as a state machine diagram.

When a new kind of rule is created any of the following hook extension points can be implemented to provide the new behaviour. Of course, some algorithm extension points could also be implemented in a plugin in order to complete the rule behaviour.

- **instantiation**. When a rule is encountered in the transformation text its body is evaluated to set up the rule properties This extension point makes it possible to set additional initialization data as if it were the rule constructor.
- **checking conformance**. Resolving a binding involves checking the conformance of the set of rules with that binding. Different kinds of rules could have different conformance strategies.

- **checking filter**. By default the rule filter is checked before applying the rule by evaluating the expression in the *filter* part. However, another kind of rule may establish another way of filtering applicable rules.
- **creating and linking**. Just before a rule is applied, new target elements are created and linked to the corresponding target feature in the binding. This behaviour could be different, for instance, rules which never transform a source element twice, i.e. no new target elements are created if the source element has been already transformed, but the previous result is linked.
- **mapping application**. Rule application normally consists of executing the mapping to assign primitive values and resolve bindings. This extension point allows a rule to modify the rule application strategy, for instance, to have more than one mapping in a rule.

Given the steps shown in Figure 3, the following *filter extension points* makes sense: after definition, before/after check condition, before/after create and link, and before/after rule application.

### 3.3 Syntax extensions

Syntax extensions allow a plugin to add new keywords and define nested structures to create new language constructs. In Section 4 we will create a *phase* syntax construct which encloses a set of rules; this construct is a new nested structure defined by a keyword such as *phase*.

There are two kinds of syntax extensions depending on the place the new keyword could appear. If the new construct may appear within a rule, it is said to be rule scoped, while if it can only appear in the transformation body, it is said to be transformation scoped. Transformation scoped syntax is intended to specify configurations that affect the whole transformation, while rule scoped syntax should be used to set rule properties.

Each new keyword is added to an associative table which associates each keyword with a callback to manage it. Such callback provides the new language construct with the proper semantics, and will be called when the keyword appears in the transformation text.

## 4 Plugin Example

In this section we will show an example of language extension whose main purpose is to allow a transformation to be organized in several phases, and thus facilitate dealing with complex transformations. We will show how this extension is useful in coping with the problem explained in Section 2 related to querying partially built target models. Also, we will show which extension points have been used to implement the phasing mechanism.

When this extension is applied to the language, a transformation is organized into several phases, where each phase consists of a set of rules which can only be invoked in the context of its phase [1], but it can query target elements which

have been partially generated in previous phases. It can be considered that each phase has a state defined by the state of all the generated target elements, both in this phase and in previous phases. A transformation progresses by applying rules which modify the current phase state. Therefore, each phase refines a partial transformation state established by previous phases. This is what we call *rule refinement*. It is worth noting that when a rule navigates on the target model, the query must be consistent with the previous phase state. Figure 4 shows how a relational model is refined in three phases: firstly data columns are created, secondly primary keys are set (marked $*$), and finally, foreign keys are created (marked $^{\rightarrow}$).

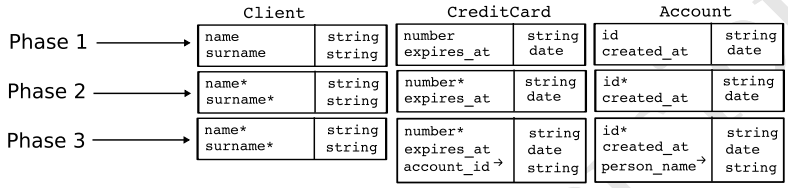


**Fig. 4.** Use of the phasing mechanism to generate a relational model by refinement.

We have identified the following requisites, which should be fulfilled by the extension:

- Rules should be enclosed by a higher level syntax structure in order to easily identify which phase a rule belongs to.
- Refinement of transformation state implies a rule can use target elements created by a rule of a previous phase.
- The execution order of phases should be specified.

To satisfy such requisites the following extension points have to be implemented. First, syntax extensions are made in the form of two new keywords in the transformation scope: `ordering` and `phase`. The *ordering keyword* makes it possible to specify the order in which phases will be applied, while the *phase keyword* expects a code block enclosing a set of rules. The example below shows what the syntax looks like.

Now, we must identify which rule and algorithm extension points have to be implemented. Considering the rule execution cycle, it is necessary to know which phase a rule belongs to. Therefore, the `after_instantiation` filter for rules must be implemented to detect when a rule has been read and within which phase.

Next, we must think about how the transformation algorithm is affected by the phasing mechanism, that is, which algorithm extension points have to be implemented. The first difference with the core algorithm is the way entry point rules are applied. Every phase should have its own entry point rules, and these are executed depending on the phase order. All this logic is implemented in the

*entry point* extension point (see Figure 2) overriding the previous logic related to the transformation start.

Finally, rule refinement must be addressed. The same rule could be defined in more than one phase but with a different mapping, and new instances are created the first time the rule is applied for a given source element, but when the rule is applied (in another phase) for the same source element, no new target elements are created and the previous ones are used to evaluate the mapping part of the rule. The convention used is that a rule with a name $x$ in a phase $A$, and a rule with a name $x$ in a phase $B$ are supposed to be the same, with $x$ of $B$ being a refinement of $x$ of $A$.

In order to do so, we need to redefine how rules are selected and evaluated to resolve bindings. First, rule selection has to take into account that only rules belonging to the current phase can be selected. Since we do not want to override the default behaviour of the `select conforming rules` extension point, but only to remove those rules not belonging to the current phase, the `select conforming rules` filter could be used. The use of this filter is a good example of reuse of previous logic (selection of conforming rules), but modifying the result to serve a new purpose. Finally, the `evaluate conforming rules` hook is overridden to keep track of rule refinement, so that new target elements are not created if they have been created by a rule in a previous phase.

Below, the class-to-table example is rewritten, applying the phasing mechanism just explained.

```
ordering :default, :primary_keys, :foreign_keys

phase 'default' do
  rule 'klass2table' do
    from  ClassM::Class
    to    TableM::Table
    mapping do |klass, table|
      table.name = klass.name
      table.cols = klass.attrs.select { |a| a.type.is_a?(ClassM::PrimitiveType) }
    end
  end

  rule 'property2column' do
    from    ClassM::Attribute
    to      TableM::Column
    mapping do |attr, column|
      column.name = attr.name
      column.type = attr.type.name
    end
  end
end

phase 'primary_keys' do
  rule 'property2column' do
    from    ClassM::Attribute
```

```
    to      TableM::Column
    filter  { |attr| attr.is_primary }
    mapping do |attr, column|
      column.owner.pkeys << column
    end
  end
end

phase 'foreign_keys' do
  rule 'klass2table' do
    from  ClassM::Class
    to    TableM::Table
    mapping do |klass, table|
      table.cols = klass.attrs.select { |a| ! a.type.is_a?(ClassM::PrimitiveType) }
    end
  end

  rule 'reference2column' do
    from    ClassM::Attribute
    to      Set(TableM::Column)
    mapping do |attr, set|
      table = klass2table(attr.type)
      set.values = table.pkeys.map do |col|
        TableM::Column.new(:name => table.name + "_" + col.name, :type => col.type)
      end
      table.fkeys = TableM::FKey.new(:cols => set)
    end
  end
end
```

As can be seen in the example, with the phasing approach, the problem of safely navigating the target model explained in Section 2 is solved. As said in Section 2, we should have chosen a more complex transformation example to show the real usefulness of phasing, but use this simpler one for the sake of clarity. The key point of this approach is that, in any phase, it can be known which target element have been already created. In the example, the `reference2column` rule can safely get all primary keys of a table because the previous phase has created them. In addition, the phasing mechanism makes it easy to deal with complex transformations requiring to be organized into more than one pass (it can be thought of as a multi-pass transformation).

## 5 Related Work

Several classifications of model transformation approaches have been developed [1][6]. According to these classifications, the different model-model approaches can be grouped into three major categories: imperative, declarative and hybrid approaches.

Some of the latest research efforts in model transformation languages are ATL, Tefkat, MTF, MTL and Kermeta. MTL and Kermeta [13] are imperative executable metalanguages not specifically intended to model-model transformation, but they are used because the versatility of their constructs provides high expressive power. However, the verbosity and the imperative style of these languages make writing complex transformations difficult because they abstract less from the transformation details and make transformations very long and not understandable.

ATL is a hybrid language with a very clear syntax [3][14]. 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.

Tefkat is a very expressive relational language which is completely usable [15]. As noted in [2], writing complex transformations in a fully declarative style is not straightforward, and the imperative style may be more appropriate. That is why a hybrid approach is a desirable characteristic for a transformation language to help in writing practical transformation definitions.

MTF [16] is a set of tools including a declarative language based on checking or enforcing consistency between models. MTF provides a extensibility mechanism to extend its syntax by plugging in a new expression language. To our knowledge, RubyTL is the first extensible model transformation language in the sense that it provides a mechanism to extend both its syntax and the transformation algorithm. In any case, the idea of extensible language has been applied in other domains [17][18], and it is widely used in the Lisp language which provides a powerful macro system to extend its syntax.

## 6 Conclusions and Future Work

MDD approach will succeed only if proper model transformation languages are available. These languages should have good properties, such as being usable and able to provide appropriate expressiveness to deal with complex transformations. In the last few years, several languages have been defined and quality requirements have been identified in proposals such as [2]. Nowadays, experimentation with existing languages is a key activity of the MDD area.

RubyTL [4] is an extensible hybrid language that provides declarative expressiveness through a binding construct. It is a usable language with a clear syntax and a good trade-off between conciseness and verbosity. Moreover, the transformation style is close to the usual background of developers.

In this paper we have described how RubyTL provides a framework to experiment with the features of the language through a plugin mechanism. When a new language feature is going to be added, a new plugin is created. Creating a plugin means identifying which extension points are involved before it is im-

plementation in the Ruby language. An example of transformation organized in phases has illustrated the process of plugin design.

The contribution of this paper is twofold. On the one hand, RubyTL is an extensible model transformation language, which provides some advantages with regard to other non extensible transformation languages: i) the language can be adapted to a particular family of transformation problems, ii) new language constructs can be added without modifying the core and iii) it provides an environment to experiment with language features. Moreover, we have proposed a phasing mechanism to allow a transformation to safely navigate the target model.

However, a limitation of our approach is that extensibility is restricted to a particular family of languages: those which rely on the binding concept. Since we have identified the extension points directly from an algorithm with "holes", only languages following such a scheme can be implemented. Anyway, the same idea can be reused to experiment with other kinds of languages. Another concern is that, since which extensions are going to be developed is not known in advance, the programmatic interface of the language should be as general as possible. At present, we are exploring ways of controlling the scope of plugins changes, so that incompatible extensions cannot be loaded at the same time.

In our experiments with the language, in addition to the phasing mechanism, we have found some language features we believe essential in this kind of transformation languages. We have identified four different types of rules, each one with some properties which help to solve certain transformation problems: *normal rules* like the ones shown in this paper, *top rules* to allow a transformation to have more than one entry point, *copy rules* which can transform a source element more than once, *transient rules* which are able to create elements that are only valid while the transformation is being executed. Moreover, a transformation language should be able to deal with one-to-many and many-to-one transformations. In this paper we have shown an example of one-to-many mappings. We are currently experimenting with different ways of implementing many-to-one mappings, since there are important performance concerns that must be taken into account. Another consideration is that, in a hybrid language, language constructs are needed to call rules explicity from imperative transformation code. Finally, an important property not found in many transformation languages is incremental consistency. We are currently experimenting with different ways to achieve it.

We continue experimenting with RubyTL by writing transformations for real problems, specifically we are applying MDD to portlets development. Also, we are currently working on the integration of our transformation engine inside the Eclipse platform by using RDT.[2] At present, an editor with syntax highlighting, a launcher for transformation definitions, a configuration tool for plugins, and a model-to-code template engine is available.[3]

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

## References


1. 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.
2. Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations, 2005.
3. 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.
4. Jesús Sánchez, Jesús García, and Marcos Menarguez. RubyTL : A practical, extensible transformation language, 2006. Sent to ECMDA'06. Available for reviewers in *http://gts.inf.um.es/files/rubytl.pdf*.
5. Dave Thomas. *Programming Ruby. The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.
6. 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.
7. 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.
8. Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, June 2005. http://www.martinfowler.com/articles/languageWorkbench.html.
9. Jim Freeze. Creating DSLs with Ruby, March 2006. http://www.artima.com/rubycs/articles/ruby_as_dsl.html.
10. Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Call for Papers of Model Transformations in Practice Workshop '05, 2005.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.
12. Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. 2001.
13. 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.
14. 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.
15. Michael Lawley and Jim Steel. Practical Declarative Model Transformation With Tefkat. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, 2005.
16. Sebastien Demathieu, Catherine Griffin, and Shane Sendall. Model Transformation with the IBM Model Transformation Framework, 2005. http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/index.html.
17. Terry A. Winograd. Muir: A Tool for Language Design. Technical report, Stanford, CA, USA, 1987.
18. Macneil Shonle, Karl J. Lieberherr, and Ankit Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA Companion*, pages 28–37, 2003.