Approaches for model transformation reuse: factorization and composition

Jesús Sánchez Cuadrado and Jesús García Molina

University of Murcia, Spain
{jesusc, jmolina}@um.es
http://gts.inf.um.es/

Abstract. Reusability is one of the principal software quality factors. In the context of model driven development (MDD), reuse of model transformations is also considered a key activity to achieve productivity and quality. It is necessary to devote important research efforts to find out appropriate reusability mechanisms for transformation tools and languages. In this paper we present two approaches for reusing model transformation definitions. Firstly, we tackle the creation of related model transformations, showing how the factorization of common parts can be achieved. Secondly, we describe a proposal on the composition of existing, separated transformation definitions so that they can be used to solve a concrete transformation problem. We illustrate both proposals with examples taken from the development of a software product line for adventure games, which has been implemented using the modularization mechanisms of the RubyTL transformation language.

1 Introduction

Reusability is one of the principal software quality factors because the reuse of software assets reduces development effort and cost, and improves quality aspects such as maintainability, reliability and correctness. In the same way, in model driven development (MDD), the reuse of model transformation definitions is also considered a key activity to achieve productivity and quality [1]. However, it is still necessary to gain more experience on transformation reuse in real projects, and to devote important research efforts to discover appropriate reusability mechanisms for transformation tools and languages.

Two points of view can be considered in the practice of software reuse: developing artifacts *for reuse* and developing new systems *with reuse* of existing artifacts. Languages and tools must support mechanisms for creating and specifying reusable artifacts, as well as mechanisms for specializing and integrating them for building a new system. Transformation reuse follows the same principles as software reuse. So, good abstractions are essential for creating reusable transformations. Current approaches are focused on reusing single rules or patterns [2][3][4]. However, practical development of large systems requires reuse to be tackled at a coarser-grained level. Transformation definitions should be reusable as a whole, and mechanisms for their specialization, integration and adaptation must be provided.

In this paper we address two model transformation reuse techniques: factorization and composition. The first deals with developing reusable transformation definitions and the second with the adaptation and integration activities. When related transformation definitions are created, they can have duplicated code. We will show how factorization of common parts in a new transformation definition, intended *for reuse*, removes the duplicated code. With regard to developing *with reuse*, we will tackle the problem of composing several, independent transformation definitions for solving a specific transformation problem. Throughout the paper we will analyze some issues related to constraints for reuse imposed by the specification of source and target metamodels in the transformation definition.We will propose solutions in the context of the RubyTL transformation language [5].

The paper is organized as follows. The next section motivates the interest in factorization and composition of model transformation definitions. Then, Section 3 introduces the example that will be used through the paper. Section 4 gives a brief explanation of the modularity mechanism provided by RubyTL. Sections 5 and 6 explain our approaches for factorization and composition in model-to-model transformations respectively. Finally, in Section 7 the related work is presented, and Section 8 presents some conclusions.

2 Motivation

Tackling large projects using MDD implies managing complex metamodels and transformations of considerable size. To deal with this complexity, decomposition in smaller parts, finding commonalities and reusing common parts are needed. Implementation of software product lines (SPL) using MDD [6] is a clear example of this situation. Model transformation languages should provide reuse mechanisms, allowing us to create transformation definitions that can be used to create different products, and that can be extended to fulfill the requirements of a concrete product.

In our experiments, which integrate software product lines and model driven development, we have had to face three problems related to transformation definition reuse.

1. *Factorizing common parts* of transformation definitions.
2. *Adapting and composing* several transformation definitions.
3. *Variability* in transformation definitions, that is, the need to attach an aspect to a transformation definition to implement a certain product variant [6]. In this paper we will not address this issue because of lack of space. In any case, we have been able to address it successfully using the phasing mechanism explained in Section 4.

In this paper, we will focus on the last two problems. Next, we set out our proposal for factorization and composition. We also introduce two key concepts in our proposal: *metamodel compatibility* and *metamodel extension*.

2.1 Transformation factorization

Transformation factorization is the process of finding common functionality shared between two or more transformation definitions, and of extracting the common parts to a base transformation definition. The non-common functionality is implemented by other transformation definitions which reuse the base transformation definition, and add their specific functionality.

Nevertheless, for a transformation definition to be reused, it is not enough that it provides the required functionality, but some kind of compatibility between metamodels of both transformation definitions must be satisfied. A transformation definition T_1 can only be directly reused within another transformation definition T_2 if each source and target metamodel of T_2 is "compatible" with the corresponding metamodel of T_1.

In Section 5, where factorization of transformation definitions is addressed using an example, we will propose an approach to deal with metamodel compatibility which relies on the notion of model type [7].

2.2 Transformation composition

As is noted in [8], transformation definitions can be composed in several ways, such as chaining definitions written in different languages (external compositon), or composing rules from two or more transformation definitions written in the same language (internal composition). Internal composition requires proper modularity mechanisms and composition operators. The composition unit can be a rule [2][4][3], or some coarser-grained construct.

Our proposal relies on a phasing mechanism [9], which is a mechanism for internal transformation composition. It is coarse-grained because it uses the concept of *phase* as composition unit, which encapsulates a set of rules aimed to perform a well-defined transformation task. The mechanism provides operators to allow transformation definitions to be composed by means of the trace information. In this paper, we will tackle the problem of composing transformation definitions whose source metamodels are the same (or at least compatible) and whose target metamodels are completely independent. Each one of the target metamodels represents a concern in the system being generated, but at some point these concerns must be connected.

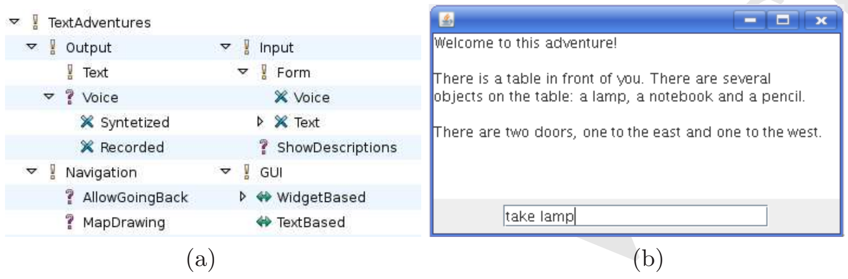
This issue arises frequently, for instance in the MDA approach when several platform specific models (PSM) are derived from the same PIM. To be able to generate a complete and meaningful architecture, the bridges between the architectural elements must be established. Our approach to solving this problem will rely on creating an *extension* of the PSM metamodels, which will be in charge of adding the metaclasses needed to establish the bridge.

We define a metamodel MM_{ext} as an *extension* of another metamodel MM_{base} when: MM_{ext} imports MM_{base} and at least one of its metaclasses is related to another metaclass of MM_{base} (either having a reference or inheriting from it).

In Section 6 we will explain the approach in detail, highlighting the problems involved at model transformation language level, and proposing a solution using the RubyTL transformation language.

3 Running example

To show the problems involved in the reuse of transformation definitions, and to illustrate our solution, we have developed a small, but non-trivial, case study. In this case study, we are interested in developing a software product line for interactive fiction games (also known as text adventures). In this kind of games, the player is presented with a text describing a situation or a room. The player interacts with the game by writing simple commands such as “get key” or “go north”. A feature model has been used to describe the game requisites and to express commonalities and variabilities in the domain. Figure 1(a) shows an excerpt of it, while Figure 1(b) shows an screenshot of the user interface.



Welcome to this adventure!

There is a table in front of you. There are several objects on the table: a lamp, a notebook and a pencil.

There are two doors, one to the east and one to the west.

take lamp

Fig. 1. (a) Simple feature model for text adventure games. (b) Screenshot of the user interface of a game generated with this SPL.

As can be seen in the feature model, a game must have input and output components (for the user to interact with the game), there is a navigation mechanism so that the user can go from one room to another (e.g. a variation must allow the user to go back to an already visited room) and a graphical user interface.

A DSL has been developed to describe a specific game, using a metamodel to represent its abstract syntax. In this DSL, the concepts involved in the description of an adventure are present, such as rooms, objects, exits, actions, etc. The example below shows a concrete syntax for the description of a room that is present in the game.

```
room 'table_and_notebook' do
  text %{ There is a table in front of you. There are
          several objects on the table: a lamp, a notebook and a pencil.
          There are two doors, one to the east and one to the west. }
  object 'lamp' do
    allowed_actions :take
    description "a normal lamp"
  end
  exit :west, :goto => 'dark_room'
end
```

The game implementation is generated automatically from this DSL using a model-to-model transformation approach. Figure 2 shows the transformation flow, where three model-to-model transformations are involved to transform the initial DSL (Adv) into the game implementation. The game architecture, independent from an implementation technology, is represented by three metamodels, each one representing a game concern. One concern is the command interface (i.e. which commands are valid in each game situation), which is represented by the Commands metamodel. Another concern is the game navigation (i.e. the mechanism in charge of moving the user from one room to another), which is represented by a state machine-like metamodel (SM). Finally, the user interface is also represented, by the GUI metamodel. These metamodels are instantiated by the adv2com, adv2sm and adv2gui model-to-model transformations respectively.

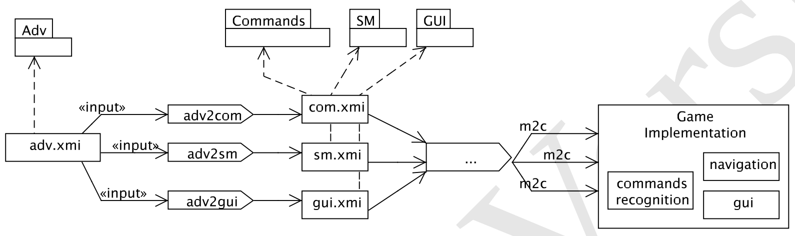


Fig. 2. Transformation flow for our software product line for adventure games. Dashed lines represents conformance relationships, while solid lines connect an input or output model with a transformation.

We will use this example product line to drive the discussion of the rest of the paper. From the example, we will identify general problems regarding transformation reuse, and we will derive general solutions for them.

4 Phasing mechanism

In [9] we presented a modularity mechanism for rule-based model transformation languages, which allows decomposition and composition of model transformation definitions. This mechanism is based on the idea of *phase*. In this section we give a brief introduction, using RubyTL as the implementation language.

With a phasing mechanism, a transformation definition is organized as a set of phases, which are composed of rules. Executing a transformation definition consists of executing its phases in a certain order. The execution of a phase means executing its rules as if they belonged to an isolated transformation definition, without conflicts with rules defined in other phases. A transformation definition is therefore seen as a phase, so allowing the same composition operators as for phases. Also, the mechanism provides a DSL to set the phase execution order explicitly.

One composition operator we have defined is a new kind of rule, called *refinement rule*, which matches against the trace information, instead of the source model. There is a match if a source instance of the metaclass specified in the rule's source pattern (i.e. rule's *from* part), has a trace relationship with one target instance of the metaclass specified in the rule's target pattern (i.e. rule's *to* part). Thus, for each match, the refinement rule is executed, but instead of creating a new target element as usual, the element matched by the target pattern is used. This means that no new target elements are created, but the rule works on existing elements, refining them.

Since a transformation definition is a form of phase, importing a separate transformation definitions within another transformation integrates seamlessly with the whole mechanism. RubyTL provides an `import` statement, which is in charge of resolving the dependency with an external definition by treating it as a phase.

The next example shows an example of phase usage. The `adv2com` transformation definition is extended to implement the `ShowDescriptions` feature, which implies creating a *describe* command for each available object in a room.

```
import 'm2m://adv2com'

phase 'show_descriptions' do
  refinement_rule 'refine_room' do
    from Adv::Room
    to   Command::CommandSet
    mapping do |room, command_set|
      command_set.validCommands = room.availableObjects
    end
  end

  rule 'obj2command' do
    from Adv::Object
    to   Command::Command
    mapping do |room, input|
      command.words << Input::Word.new(:value => 'describe')
      command.words << Input::Word.new(:value => object.name)
    end
  end
end

scheduling do
  execute 'adv2input'
  execute 'show_descriptions'
end
```

First of all, the `adv2com` definition is imported, so that it can be scheduled as a normal phase. Secondly, the `show_descriptions` phase contains a refinement rule that refines the `Room` to `CommandSet` mapping, so that the set of valid commands is extended. Note that no new `CommandSet` elements are created, but the match is against those `Room` objects that are related by the trace to already created `CommandSet` objects. Finally, the `scheduling` block is in charge of setting the order in which phases are executed.

5 Transformation factorization

In this section, we will show how to factorize common parts of two transformation definitions into a base transformation, which is then reused so that code duplication is avoided.

In the game, each room has an associated set of valid commands the user can issue, which are derived by the `adv2com` transformation. It creates the `com.xmi` model, conforming to the `Commands` metamodel, which represents an implementation independent view of a command-based interface. From this model, a concrete implementation must be generated.

The implementation of the `Input` feature (see Figure 1(a)) requires dealing with different technologies to handle whether speech or written text is used to enter commands to the game. A model-to-model transformation approach has been used to tackle this issue. Commonalities in speech and text recognition technologies have been studied in order to reuse as many transformation definitions and metamodels as possible.

In particular, we have detected commonalities in the way the text or speech structure is recognized. Some speech recognition technologies use a special kind of grammar (JSGF, Java Speech Grammar Format), which is very similar to an EBNF grammar, but adding special constructs, such as the possibility of attaching a weight to a grammar element. Moreover, EBNF grammars are classic artifacts for representing text recognizers.

A metamodel to represent an EBNF grammar has therefore been created. In order to depict JSGF grammars, an extension of the former metamodel has also been created. As explained in Section 2.2 this means that the JSGF contains metaclasses which inherit from metaclasses defined in the EBNF metamodel (Figure 3(a) shows an excerpt of both metamodels and their inheritance relationships). The `text` feature is implemented as a model-to-model transformation from the `Commands` metamodel to `EBNF` (`com2ebnf`), while the `speech` feature is implemented as another model-to-model transformation from the `Commands` metamodel to `JSGF` (`com2jsgf`). Since JSGF has been modelled as an extension of EBNF, both transformation definitions share most of the functionality. Our approach for *factorizing* the common parts of the transformations relies on creating a base transformation, which is imported by the concrete transformations that implement the non-common functionality. This schema is illustrated in Figure 3(b).

The `com2jsgf` transformation outputs models conforming to both JSGF and EBNF metamodels (i.e. JSGF is an extension of EBNF, so JSGF models can be used where EBNF models are expected), and it reuses `com2basic`, whose output model must conform to the EBNF metamodel. The constraint for reuse regarding metamodel conformance is that the metamodels used by one transformation provide the classifiers expected by the reused transformation, that is, metamodels must be compatible in some way.

To formalize this constraint we rely on the model type concept. In [7] a formalization of model types is presented, where the type of a model is seen as

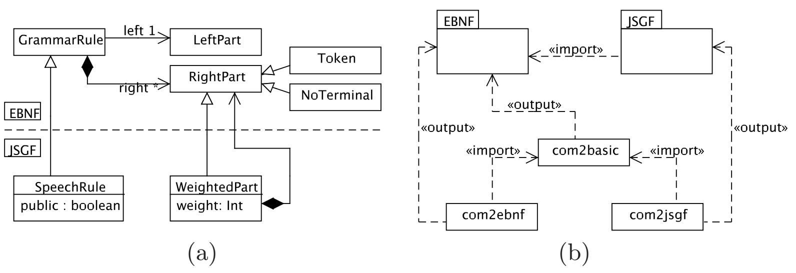


Fig. 3. (a) Excerpts of the EBNF and JSGF metamodels. (b) Relationships between reused transformations and their metamodels.

the set of the types of all objects belonging to the model. This definition allows models to conform to several metamodels.

At this point, we will address the compatibility problem posed in Section 2.1. From a transformation language point of view, the type of a model is the set of classifiers that are used within the transformation. In the same way as the previous definition, a transformation input or output model can conform to several metamodels. However, the problem arises when two metamodels contain a classifier with the same name, because the name clash prevents the transformation language from knowing which of them is being referred to by the name.

Usually, transformation languages define one namespace for each input or output model, which is bound at launch time to the concrete metamodel, so that classifiers are referenced through the corresponding namespace. If a metamodel imports another metamodel, when it is bound to the namespace the transitive closure on classifiers can be made to give access to the imported metamodel's classifiers. However, this approach has two problems: (1) it does not prevent name clashes, and (2) models conforming to several non-related metamodels cannot be handled.

We propose allowing several namespaces to be defined for each model, reflecting the fact that the model conforms to several metamodels, so that each metamodel is handled by its corresponding namespace. Thus, when a transformation is going to be launched these bindings must be established in some way. In our tooling [10] we use a DSL to set the models to be used in a transformation execution, and to bind the namespaces with the metamodels. The piece of DSL below shows how the concrete metamodels (`jsgf.ecore` and `ebnf.ecore`) are bound to the namespaces `EBNF` and `JSGF` expected by the transformation definition. Moreover, since the `com.xmi` model conforms to only one metamodel, a classic approach can be used (i.e. only one namespace per model).

```
model_to_model :com2jsgf do
  sources :namespace => 'Commands',
          :metamodel => 'commands.ecore',
          :model     => 'com.xmi'
  targets :model => 'jsgf.xmi',
          :namespaces => {
            'JSGF' => 'jsgf.ecore'
            'EBNF' => 'ebnf.ecore'  }

  transformation 'm2m://com2jsgf.rb'
end
```

In this way, the transformation definition that creates a JSGF model to implement speech recognizers (`com2jsgf` in Figure 2) expects a metamodel to be bound to the `Commands` namespace, while it expects the `EBNF` and `JSGF` target namespaces to be bound too.

Below, an excerpt of the `com2jsgf` transformation definition is shown. It reuses the `com2basic` transformation definition using an `import` statement as explained in Section 4. Since the `com2basic` transformation definition may expect different namespace names to be bound (e.g. it may name the target namespace as `EGrammar` instead of `EBNF`), the import statement must provide a renaming facility. In our implementation, the binding between the current namespaces and the ones expected by the reused transformation is made using the `map` statement.

```
transformation 'com2jsgf'
source 'Adv'
target 'EBNF', 'JSGF'

import 'm2m://com2basic' do
  map 'Commands' => Commands
  map 'EGrammar' => EBNF
end

phase 'voice2grammar' do
  ... Transformation rules creating JSGF and EBNF elements...
end

scheduling do
  execute 'basic-grammar'
  execute 'voice-grammar'
end
```

It is also worth noting that the transformation definition does not know whether it receives one model conforming to two metamodels, or two models conforming to one metamodel each. Also, it does not specify a specific version of the metamodel, but different versions of the metamodel are allowed (for instance, several versions of UML can be handled). All these versions must have at least the classifiers used in the transformation rules, and the classifiers' properties used within the rules. A formalization on these constraints is made in [7].

6 Transformation composition

This section presents an approach for composing independent transformation definitions, so that their relationships can be established. We illustrate it with an example intended to bridge the game concerns introduced in Section 3.

As explained in Section 3, we have defined three separate model transformations to instantiate each of the three concerns of our game architecture. These concerns, depicted as the `Commands`, `SM` and `GUI` metamodels, are completely independent of each other (i.e. they do not have references between them). Thus, they can be reused in contexts different from this transformation flow (e.g. a command-based input interface can also be used in command-line applications).

However, since each part of the game must interact with other parts, the three transformation definitions need to be composed to generate the bridges between the different parts. As a result of this composition, model elements of different metamodels must be connected. Typically, a connection between a model element a_1 and another model element a_2 implies that either the a_1's metaclass or the a_2's metaclass contains a reference to the other metaclass. This problem will always arise when we need to relate elements that belong to independent metamodels, as occurs in this case.

The challenge is to reuse the metamodels and their associated transformations, while being able to establish connections between them. We propose to extend the metamodels which must contain the connections, creating a new metamodel where, for each metaclass that needs to be related with another metaclass, a new metaclass inheriting from it is created. This new metamodel represents the connections between concerns in the architecture, and is usually small, because it only defines the minimum number of metaclasses to establish the connections. Another approach would be to use model weaving techniques [11] to represent these relationships, but the problems involved at model transformation level would be the same.

Figure 4 shows our approach to create a new metamodel (`ExtSM`) which extends the original `SM` metamodel to bridge states with the set of available commands for this particular game state. A subclass of `SM::State` is created, also called `State`, which has a reference to the `CommandSet` metaclass in the `Command` metamodel.

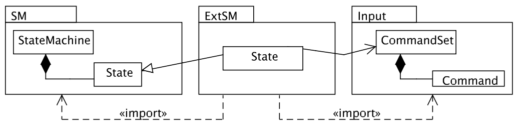


Fig. 4. Schema for creating a new metamodel bridging the `SM` and `Command` metamodel.

Once metamodels are integrated, transformations need to be integrated as well, so that bridges can be instantiated. Our strategy relies on importing and executing the transformation definitions that are going to be bridged in the context of another definition, which is actually in charge of locating the join points and instantiating the connections. It is important to notice that each transformation is imported as a black-box, which creates a piece of target model as a result, that must be integrated with the other transformation execution results. Since the interface of a transformation definition are the mappings it defines (i.e. its rules are "private"), rule-level composition mechanisms such as rule inheritance [2][4] or pattern factorization [3] cannot be applied.

According to the scheme shown in 2, the `adv2sm` transformation instantiates `State` objects conforming to the `SM` metamodel. However, we need to have `State` objects from the `ExtSM` metamodel instantiated by `adv2sm`, but its transformation rules have fixed metaclasses in their *to part*. This problem is similar to that of object oriented languages, where object creation statements using concrete classes limit reusablity and extensibility.

We propose to tackle this issue using an approach inspired by the factory patterns [12]. RubyTL allows a hook to be specified in an instantiation statement (typically the `to` part of a rule). The hook definition takes a base metaclass, which is used as the default metaclass in the case of the hook never being filled. When the transformation is imported into another transformation module, such a hook can be filled with the concrete metaclass to be instantiated, which must be a child metaclass of the base metaclass specified in the hook definition. The following piece of transformation shows the definition of a hook on the `room2state` rule: the hook is called `vstate` and its base class is `SM::State`.

```
top_rule 'sm' do
  from Adv::Adventure
  to   SM::StateMachine
  mapping do |adv, machine|
    machine.name   = adv.name
    machine.states = adv.rooms
  end
end

rule 'room2state' do
  from Adv::Room
  to   hook(SM::State => :vstate)
  mapping do |situation, state|
    state.name = situation.name
  end
end
```

Once the problem related to model elements instantiation has been solved, the next step is to actually integrate the transformation definitions. As explained above, we propose to define an "integration" transformation definition , which is in charge of composing and adapting the `adv2sm`, `adv2com` and `adv2ui` transformation definitions. It imports these three transformation definitions, and the integration is performed in three steps:

1. The target metamodel for state machines is an extension of the original one (as shown in Figure 4).
2. The `adv2sm` is parametrized by means of the `vstate` hook, so that the type of states can vary.
3. The "integration" transformation locates the join points using a `refinement rule` to match against the trace information (e.g. any *state* created from the same *room* as a another *command set* must be bridged). Then, new elements are created (either by other rules or imperatively) to adapt the existing model elements.

The following piece of code corresponds to our implementation in RubyTL of the "integration" transformation definition.

```
transformation 'integration'
source 'Adv'
target 'SM', 'ExtSM', 'Commands', 'GUI'

import 'm2m://adv2com'
import 'm2m://adv2ui'
import 'm2m://adv2sm' do
  map     'SM'    => SM
  factory :vstate => ExtSM::State
end

phase 'merge' do
  refinement_rule 'merge_rule' do
    from Adv::Room
    to   ExtSM::State, Comands::CommandSet, UI::CompositePanel
    mapping do |situation, command_set, state, ui|
      cconnector            = ExtSM::CommandConnector.new
      cconnector.commandSet = command_set
      state.input           = cconnector

      gconnector            = ExtSM::GUIConnector.new
      gconnector.gui        = ui
      state.ui              = bconnector
    end
  end
end

scheduling do
  execute 'adv2input'
  execute 'adv2sm'
  execute 'adv2ui'
  execute 'merge'
end
```

The third import statement binds `SM` namespace in the `adv2sm` transformation with the current `SM` namespace. Moreover, it parametrizes `adv2sm` so that instead of creating `SM::State` objects it creates `ExtSM::State` objects. Notice that although `adv2sm` has been bound to use the `SM` metamodel, it can create model elements conforming to `ExtSM::State` because such a metaclass is a subclass of `SM::State`.

Finally, the refinement rule is in charge of bridging related elements, creating the proper connector objects imperatively, and linking them to the state. It is worth noting that this rule matches to existing target elements, which are related to the same source element by the trace information. In this way, the join point to weave the elements is defined based on the trace information.

7 Related work

We have illustrated this paper with an example product line which has been implemented using the approach proposed by Voelter in [13]. In this proposal, variability in transformation is a key issue. Although we have not explained in detail how to implement variability in transformations, the mechanism explained Section 4 can be used for this purpose.

In [14] an infrastructure for defining model transformation chains is presented. It studies the problems involved in reusing transformations written in different languages within a transformation chain. However, it does not propose any concrete mechanism to compose transformation definitions, although it mentions as future work relying on trace information to achieve composition.

Transformation reuse at rule-level has been addressed in several works. In [2], an study of rule-based mechanisms to modularize transformation definitions is presented. Transformation languages such as Tefkat [4], ATL [15], Viatra [3] or QVT[16] provide mechanisms for fine-grained reuse, that is, reuse of rule definitions. On the other hand, reuse at a coarse-grained level has not been extensively treated. In [17] an approach based on the transformation pattern is presented. Instead of establishing constraints on the source and target metamodels, transformations are created to adapt models to the expected metamodel. Higher-order transformations is also a means to tackle transformation reuse [15][18]. Regarding model transformation languages and their mechanisms for coarse-grained reuse, we have compared RubyTL with oAw's Xtend [13] and ATL [15].

Xtend is an imperative model transformation language, based on transformation functions, which includes explicit support for transformation aspects. This mechanism is suitable for factorizing common transformation code but not for composition of transformations using the strategies explained in 6.

The ATL language provides a facility called *superimposition* which allows several transformations to be superimposed on top of each other, yielding a final transformation containing the union of all transformation rules and helpers. It is a white-box mechanism, like a form of copy-paste, which is well-suited to reusing rules within related transformations, especially when combined with rule inheritance. On the other hand, it is not a good mechanism for composing independent transformations, since they would need to expose their implementation.

In ATL, naming conflicts are solved by adding the metamodel package name as part of the classifier's name. The problem of this approach is that it does not permit metamodel variation [14], since it relies on the package name, making it impossible to use different versions of the same metamodel.

Finally, regarding QVT, it provides reuse mechanisms at rule level, such as rule inheritance [16], but it does not provide any coarser-grained composition mechanism. Also, with respect to the notion of model type explained in Section 2, our approach coincides with that adopted by the QVT specification (in particular with effective type conformance). However, we propose to allow a model to conform to several metamodels, while in QVT a model can conform only to one metamodel.

8 Conclusions and Future Work

Model transformation reuse is an important issue for model driven development to succeed when applied to large projects. Techniques and constructs at model transformation level are needed for abstracting, selecting, specializing and integrating reusable transformation definitions.

In this paper, we have presented two approaches intended to factorize and compose transformation definitions. Through a running example we have shown the problems involved and how to solve them in RubyTL. Anyway, the approaches are applicable to other languages. Beyond addressing factorization and composition of model transformations, other important contributions of this work are:

– We have shown the constraints regarding metamodels and transformation reuse, and we have tackled these contraints using the idea of model type.
– We have given a proposal to make model element creation independent of the concrete types set in transformation rules, so that a black box approach for transformation composition can be achieved.
– We have illustrated our approach with a small, but not trivial implementation of a software product line using MDD. It is available for download at *http://gts.inf.um.es/age*.

Regarding future work, an interesting issue to be studied is whether it is possible to achieve external transformation composition using these approaches. We are studying how to "connect" different languages by means of trace information.

Acknowledgments

This work has been partially supported by Fundación Seneca (Murcia, Spain), grant 05645/PI/07. Also, we would like to thank pure-systems for supporting us by letting us use its variability managment tool: pure::variants.

References

1. Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
2. Ivan Kurtev, Klaas van den Berg, and Frédéric Jouault. Rule-based modularization in model transformation languages illustrated with ATL. *Sci. Comput. Program.*, 68(3):111–127, 2007.
3. András Balogh and Dániel Varró. Pattern composition in graph transformation rules. In *European Workshop on Composition of Model Transformations*, Bilbao, Spain, July 2006.
4. Michael Lawley and Kerry Raymond. Implementing a practical declarative logic-based model transformation engine. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 971–977. ACM, 2007.
5. Jesús Sánchez, Jesús García, and Marcos Menarguez. RubyTL: A Practical, Extensible Transformation Language. In 2^{nd} *European Conference on Model Driven Architecture*, volume 4066, pages 158–172. LCNS, June 2006.
6. Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.
7. Jim Steel and Jean-Marc Jzquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):452–468, December 2007.
8. Anneke Kleppe. MCC: A model transformation environment. In 2^{nd} *European Conference on Model Driven Architecture*, pages 173–187. LCNS, 2006.
9. Jesús Sánchez Cuadrado and Jesús García Molina. A phasing mechanism for model transformation languages. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1020–1024, New York, NY, USA, 2007. ACM Press.
10. Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *IEEE Softw.*, 24(5):48–55, 2007.
11. Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.
13. Markus Voelter and Iris Groher. Handling variability in model transformations and generators. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM07)*, 2007.
14. Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Uniti: A unified transformation infrastructure. In *Proceedings of the MoDELS 2007 Conference*, volume 4735 of *LCNS*, pages 31–45. Springer, 2007.
15. Fréderic Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.
16. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005. www.omg.org/docs/ptc/05-11-01.pdf.
17. Edward D. Willink and Philip J. Harris. The side transformation pattern: Making transforms modular and re-usable. *Electr. Notes Theor. Comput. Sci.*, 127(3):17–29, 2005.
18. Jon Oldevik and Oystein Haugen. Higher-order transformations for product lines. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 243–254, Washington, DC, USA, 2007. IEEE Computer Society.