A phasing mechanism for model transformation languages

Jesús Sánchez Cuadrado
University of Murcia
Murcia, Spain
jesusc@um.es

Jesús García Molina
University of Murcia
Murcia, Spain
jmolina@um.es

ABSTRACT

In recent years a great effort has been devoted to understanding the nature of model transformations. As a result, several mechanisms to improve model transformation languages have been proposed. Phasing has been proposed in some works as a rule scheduling or organization mechanism, but without any detail.

In this paper, we present a phasing mechanism a we explain in detail how it can be integrated in a transformation language and when its usage is appropriate. The mechanism we propose can be seen as an internal transformation composition mechanism. First, we motivate the work, and then we describe in depth the mechanism. Finally, we show four examples of application and give some conclusions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*modules, packages*; D.3.2 [**Language Classifications**]: Specialized Application Languages—*model transformation languages*

General Terms

Design, Languages

Keywords

Model transformation, transformation languages, phasing mechanism, transformation definition modularity, internal transformation composition

1. INTRODUCTION

In the last four years a considerable number of model transformation languages have been developed both in the academic field and in software companies, and even from open source projects. The OMG has proposed the QVT language as a standard, but the success of QVT is not clear today. All these languages have allowed transformation definitions to be written and some knowledge has been acquired on the nature of transformations. However, model transformations are not well understood yet, and it is necessary to continue the efforts to write transformations for real problems.

In this paper we will present a phasing mechanism to organize a model transformation definition in several phases, and we will explain when and how to use it. Moreover, we will show some advantages of applying the phasing approach. In the feature model proposed in [2], phasing is considered a rule scheduling mechanism: "the transformation process may be organized into several phases, where each phase has a specific purpose and only certain rules can be invoked in a given phase", but the authors do not provide more details.

A phasing mechanism is also mentioned in other works such as [1][7][3]. In OptimalJ [1] the transformation process is organized in a fixed number of phases, and the user has no control over the phasing mechanism. In [7], Warmer presents phasing as a way of composition in the small "to give the user control on the overall transformation" and to provide "separation of concerns". In [3], Kurtev proposes a language, Mistral, with a mechanism to organize a transformation in several steps, but no further details are given. These works point out that phasing is a feature worth being supported by a model transformation language, but the mechanism is superficially described.

In [5] a phasing approach intended to solve the problem of reading partially built target models is described. This paper also describes an implementation of the mechanism as a plugin for the RubyTL language [6]. Although our proposal comes from the approach explained in [5], we have changed the syntax and semantics of the language constructs related to the phasing mechanism in order to avoid a tight coupling between phases. Thus, with our approach it is possible to obtain a certain degree of modularity in the sense of [3], as we will explain in Section 3. Therefore, the phasing mechanism we propose can be seen as an internal transformation composition mechanism, since it deals with composing rules of a transformation executed by a single tool.

The paper is organized as follows. In this section we have motivated our proposal. Section 2 describes the proposed phasing mechanism. Section 3 shows how and when to use phasing, through some examples, and finally Section 4 presents the conclusions.

2. PHASING MECHANISM

In this section we will explain the mechanism we propose to organize a transformation definition in several phases, where each phase is composed by one or more rules. This mechanism can be understood as a modularity mechanism since phases can be composed so that a complex transformation can be built up from smaller transformations [7], or the other way around, a complex transformation can be decomposed into a set of less complex transformations (a phase can be seen as a simpler transformation). In this explanation we will not focus on any concrete transformation language, but the examples proposed in Section 3 will be written in RubyTL [6].

2.1 Mechanism structure

The abstract syntax corresponding to the phasing part of a transformation language is shown in Figure 2.1. With a phasing mechanism, a transformation definition is organized in an ordered set of phases and each phase consist of a set of rules that will be executed by the transformation engine to perform a certain task. In addition, a phase has a conditional expression to prevent its being applied if the condition is not held by the source or target models. This condition can be seen as a phase precondition. It also worth noting that phases are ordered within a transformation, since the order in which phases are executed is decided by the user, not by the transformation engine (this is because phasing can be seen, to some extent, as a scheduling mechanism). Finally, a phase can be switched on/off, as the *switchedOn* attribute expresses. To allow a phase to be properly switched off, the *depends* relationship establishes the dependence between phases, as will be explained in Section 2.2

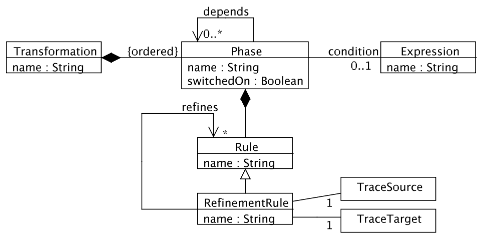

Figure 1: Part of the abstract syntax of a language providing phases.

Now, we will introduced the notion of *transformation state* that will be used throughout the paper. A transformation has a state defined by the state of all generated target elements (i.e. the current state of the target model), and the execution of a phase changes the current state by applying the rules. Therefore, each phase modifies a partial transformation state established by previous phases.

As can be seen in the abstract syntax of Figure 2.1, there is a special kind of rule, called "refinement rule" that can be related to one or more rules belonging to previous phases through a relationship called *refines.* This relationship means that a refinement rule refines the work done by one or more rules in a previous phase. A refinement rule has the same structure as a normal rule, but instead of a source pattern and a target pattern, it has a `trace_source` pattern and a `trace_target` pattern which match against the trace information, instead of the source and target models. We call this feature *transformation refinement* and it is explained further in Section 2.2.

Each phase defines a new rule scope, that is, it is possible to have a rule with the same source pattern and the same target patten in two different phases without any collision because the execution context in which such rules are evaluated is different. The following example shows two phases each containing a rule with the same source and target pattern.

```
phase example-phase-1
  rule a-to-bc
    source A
    target B, C

phase example-phase-2
  condition { A.allInstances->size < 10 }
  rule a-to-bc
    source A
    target B, C
```

As the example shows, these rules are completely independent due to the fact that each phase defines a new rule scope. This property is useful to allow a rule to act over the same elements but behaving differently according to the current transformation state, that is, depending on the phase it belongs to. In the example, the rule `a-to-bc` in the first phase creates one element of type `B` and one element of type `C` for each source element of type `A`. The second phase is only executed when the condition is true, in this case, whenever there are less than ten elements of type `A`. In this phase, the rule `a-to-bc` has the same source and target patterns as the rule `a-to-bc` in the previous phase, but there is no conflict because they are in a different rule scope. Thus, new elements of type `B` and `C` are created. It is important to note that, the rule `a-to-bc` in the first phase and the rule `a-to-bc` in the second phase are not the same rule, since they belong to different scopes.

2.2 Mechanism behaviour

The basic behaviour of the mechanism can be explained without considering a concrete transformation language, but the transformation refinement capability is dependent on the kind of transformation language.

The execution mechanism of phases is very simple. Since phases are ordered within a transformation, they are executed in the same order as they appear in the transformation definition (in any case, another mechanism can be used to set the phase ordering, for instance the dependences between phases can be used to set a correct phase ordering). If a phase precondition does not hold when the phase is going to be executed, then the concrete implementation of the phasing mechanism must choose whether to continue without executing such a phase or to stop the transformation execution. The execution of a phase means executing the rules enclosed in such a phase and, since each phase defines a new rule scope, the transformation engine could execute the phase as if it were an isolated transformation definition.

As explained in Section 2.1 a phase can be switched on/off, and a phase dependency tracking exists to prevent switching off a phase on which other phases depend. For instance, in a UML to Java transformation organized in phases, if the user does not want to generate accessor methods, he or she only needs to switch off the right phase. On the other hand, the dependence mechanism will prevent the user switching on the phase to generate accessor methods if the phase to generate instance variables has been switched off.

The key point of the phasing mechanism is *transformation refinement*. The purpose of this feature is to refine the current transformation state by allowing a rule to continue working on the target elements created by the rules of previously executed phases. Thus, the phasing mechanism execution is not simply executing a sequence of phases, but should provide a way to allow a phase to explicity refine the work done in previous phases (i.e. modifying existing target elements). This can be seen as the composition operator for phases.

There are several ways to provide such a composition operator. The simplest way is to explicity use the name of the rules whose results will be refined [5], but this approach causes a tight coupling between phases. To avoid this, our approach relies on the internal transformation trace, usually kept by the transformation engine. The transformation trace contains the information about which target elements have already been created, from which source elements and by which rule. Since we want to refine existing target elements, the trace information is queried to perform a match and work on it.

To accomplish this, we have defined a special kind of rule which is called *refinement rule*, as mentioned above. This kind of rule has a `trace_source` and `trace_target` pattern which match against the trace, instead of against a source and target model as normal rules do. It is woth noting that with this approach, the *refines* relationship is no longer made explicit by any concrete syntax, but it is implicit in the definition of the `trace_source` and `trace_target` patterns.

The match between the source and target patterns and the trace is performed in the following way. For each instance of the metaclass (including instances of subclasses) specified in the `trace_source` pattern there is a match if there exists an instance of each of the metaclasses (including subclasses) specified in the *trace_target* pattern which have been created from the same source instance. This means that a trace from the source instances to each one of the target instances must exist .

For each match, the refinement rule is executed, but instead of creating new target elements as usual, the elements matched by the *trace_target* pattern are used. This means that no new target elements are created, but the rule works on existing elements, refining them. Therefore, with this approach, phases are loosely coupled as there is no need to make an explicit reference to any rule of another phase. In addition, a phase does not need to know which phase or phases have created the trace (i.e. it is not necessary to distinguish between the traces created by each phase), since a phase only requires that a certain trace exists, regardless of the phase from which it has been created.

The following example shows the behaviour of a refinement rule. The first phase creates the elements `b1`, `b2` and `b3` of type `B`, which are related to the elements `a1`, `a2` and `a3` of type `A` in the trace, as shown in Figure 2.2. In the second phase, new target elements `c1` and `c2` of type `C` are created from source elements of type `A`, and they are also recorded by the internal traceability mechanism. The third phase has a refinement rule which is in charge of refining the work done by the previous phases. This rule refines all elements of type `B` and type `C` which are related, through the trace, to the same element of type `A`. Besides, since it is a refinement rule, no new targets elements are created, but the previously created target elements are refined.

Given the trace of Figure 2.2, the rule `a-to-bc` will be applied twice: once for {`a1, b1, c1`}, and once for {`a2, b2, c2`} such that the target elements are refined. However, the elements {`a3, b3`} are not refined since they are not related to any element of type `C`.

```
phase example-phase-1
  rule a-to-b
    source A
    target B

phase example-phase-2
  rule a-to-c
    source A
    target C

phase example-phase-3
  refinement_rule a-to-bc
    trace_source A
    trace_target B, C
```

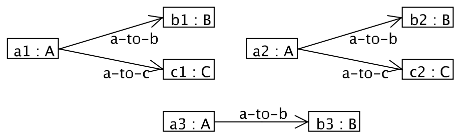

Figure 2: Transformation trace example. Source and target elements are related by an arrow with the name of the rule which performed the mapping.

3. USEFULNESS OF THE APPROACH

In this section we will show four situations where the phasing approach can be applied.

3.1 Writing complex transformations

The most obvious application of this mechanism is to reduce the complexity of a transformation by decomposing it into several steps, each one dealing with a certain part of the whole transformation.

In this way, the developer can map his or her mental scheme directly into the transformation definition: first a task is performed, then a second task which depends on the previous one, and so on. The idea is to perform a transformation by refinement steps.

OptimalJ is an example of a tool which decomposes a transformation into phases to deal with complexity, but the mechanism we propose is more general than the one used in OptimalJ where there are just three fixed phases, preventing the user controlling the transformation process. Our proposal allows a flexible number of phases to be defined (in the sense explained in [7]), such that each transformation definition can define as much phases as needed. Besides, OptimalJ is a structured-oriented and target-driven approach [2][1], while our approach can be applied to source driven languages that rely on a more flexible rule organization.

3.2 Reading target models

Another application of the approach is to allow the transformation language to read the target model safely. Usually, in a transformation language reading the target model is not safe (or even is not permitted), since the internal scheduling mechanism of the language cannot ensure that the target elements are available when they are going to be read. In particular, this is true when there exist circular dependences between the target elements. The same principle applies in the case of reading the trace information in runtime.

When a transformation is organized in phases, each phase may deal with creating some part of the target model, that is, modifying a certain part of the transformation state. Thus, when a a phase is executed, it can safely refer to a previous transformation state (i.e. the part of the target model that has been already generated). This means that it is safe to read part of the target model and to read the trace information. This is always true if the query (either on the target model or the trace) is consistent with the current transformation state. What the phasing mechanism provides is a way to ensure that queries are consistent with the current transformation state, since it is always safe to refer to target elements in a refinement rule. In addition, a tool can analize the transformation phases and determine whether the queries are consistent or not.

An example of this situation appears in a transformation between a class model and a relational model. If primary keys of a table are used to compute the foreign key columns of another table which refers to the former, then if the source model has circular dependence between classes, it may happen that primary keys are not available when needed. An example addressing this problem can be found in [5] .

3.3 Improving modularity

To illustrate how the phasing mechanism can improve modularity of transformations, we will use the example of the *examination assistant* explained in [4] where problems related to scattering and tangling are identified in a transformation between a class model for exams and questions and a user interface model based on the *Model-View-Controller* (MVC) pattern. Exam questions are represented by the `ExamElement` abstract class and there are two concrete kinds of exam questions: `OpenElement` and `MultipleChoiceElement` (they are both subclasses of the former class). Regarding the MVC pattern, there are three abstract classes: `Model`, `View`, `Concrete`. Thus, for each kind of MVC class there are concrete subclasses.

The transformation shown below is organized in phases and it solves the scattering and tangling problems by applying the transformation refinement mechanism explained above.

```
phase 'model' do
  rule 'OpenQuestionModel' do
    from XML::OpenElement
    to   MVC::Open
  end

  rule 'MultipleChoiceModel' do
    from XML::MultipleChoiceElement
    to   MVC::MultipleChoice
  end
end

phase 'view' do
  rule 'OpenQuestionView' do
    from XML::OpenElement
    to   MVC::OpenView
  end

  rule 'MultipleChoiceView' do
    from XML::MultipleChoiceElement
    to   MVC::MultipleChoiceView
  end
end

phase 'controller' do
  rule 'OpenQuestionController' do
    from XML::OpenElement
    to   MVC::OpenController
  end

  rule 'MultipleChoiceController' do
    from XML::MultipleChoiceElement
    to   MVC::MultipleChoiceController
  end
end

phase 'mvc' do
  refinement_rule do
    from XML::ExamElement
    to   MVC::Model, MVC::View, MVC::Controller
    mapping do |element, model, view, controller|
      model.view      = view
      view.controller = controller
    end
  end
end

phase 'layout' do
  refinement_rule do
    from XML::ExamElement
    to   MVC::View
    mapping do |element, view|
      view.fontName = 'Times'
      view.color    = 'Red'
    end
  end
end
```

To avoid *tangling* each phase only deals with one concern, namely the first phase deals with the model, the second phase deals with the view and the third phase deals with the controller. Thus, these phases are completely independent and they can evolve individually. The fourth phase is in charge of weaving the three concerns. To achieve this, the refinement mechanism is used to refine the result of all the rules in the previous phases. Notice that `ExamElement` is a supertype of both `OpenElement` and `MultipleChoiceElement` which are the source metaclasses of the rules to be refined. In addition, `Model`, `View`, `Controller` are also related by a subtyping relationship with the target metaclasses of the refined rules (actually all rules of the three first phases).

In the case of *scattering* the rules `OpenQuestionView` and `MultipleChoiceView` are also refined, so the layout assignments are not scattered through view rules but in a final phase.

3.4 Giving user control over the transformation

Organizing a transformation in phases is also a way to give the user control over which parts of the transformation will be executed by switching on/off some phases, as stated in [7]. In fact, the implementation of these parts can be changed or replaced if the transformation language offers some mechanism, such as transformation inheritance or superseding.

The following example is a simple transformation between an UML class model and a Java model, performed in two phases. The user can switch on/off the second phase depending on whether he or she wants a lazy initialization of instance attributes in the *get* methods or not.

```
phase 'classes-and-attributes' do
  rule 'class2javaclass' do
    from UML::Class
    to   Java::Class
    mapping do |klass, javaclass|
       javaclass.name       = klass.name
       javaclass.attributes = klass.attributes
       javaclass.methods    = klass.attributes
    end
  end

  rule 'attribute2get' do
     from UML::Attribute
     to   Java::Method
     mapping do |attr, method|
       method.name = 'get' + attr.name.capitalize
       method.type = attr.type
       method.body = 'return ' + attr.name
     end
  end
end

phase 'lazy-evaluation' do
  depends_on 'classes-and-attributes'
  refinement_rule do
    trace_from UML::Attribute
    trace_to   Java::Method
    filter { |attr, method| method.name =~ /^get/ }
    mapping do |attr, method|
      method.body = 'if ( #{attr.name} == null )' +
        'this.#{attr.name} = new #{attr.type}()' +
        method.body
    end
  end
end
```

The first phase maps UML classes into Java classes and it creates *get* methods for each attribute. The second phase refines the first phase by transforming the previously created *get* methods to support lazy initialization of attributes (adding the *if* statement to check if the element has already been created). It is worth noting that the refinement rule expects to refine *get* methods (as its filter shows) without taking into consideration which rule creates them.

4. CONCLUSIONS

In this paper we have described a phasing mechanism which can be used to deal with the complexity of model transformations in several ways, such as decomposing a transformation into several steps, being able to read a partially built target model in a consistent manner, or improving modularity of transformation definitions.

Furthermore, we have been able to test if our approach is feasible by creating a prototype implementation as a plugin for RubyTL[1].

However, there are still some questions about the mechanism to be looked into. We have shown a general overview of how a phasing mechanism works, and we have shown examples in a concrete language, but it is still an open question how much the semantics of other languages, for instance QVT, need to be changed to support this mechanism. Another concern worth mentioning is if there are other ways of composing phases in addition to using the internal trace of the transformation engine, as we have explained.

An interesting issue to be studied is whether it is possible to use the phase composition mechanism based on traces as an external composition mechanism, instead of using it as a way of composing phases written in a single language. We would like to address the problem of interoperability between transformation languages (even when they belong to different paradigms) by providing a common interface to the transformation trace. Thus, it would provide a mechanism to implement a single transformation with several languages, whose transformation engines would perform the transformation execution relying on the same trace information.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] Compuware. OptimalJ, 2005. Available online: http://www.compuware.com.

[2] K. Czarnecki and S. 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.

[3] I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005. ISBN 90-365-2184-X.

[4] I. Kurtev, K. van den Berg, and F. Jouault. Rule-based modularization in model transformation languages illustrated with ATL. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC'06)*, pages 1202–1209, Dijon, France, 2006. ACM Press.

[5] J. Sánchez and J. García. A plugin-based language to experiment with model transformations. In *9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199, pages 336–350. Lecture Notes in Computer Science, October 2006.

[6] J. Sánchez, J. García, and M. Menarguez. RubyTL: A Practical, Extensible Transformation Language. In *2nd European Conference on Model Driven Architecture*, volume 4066, pages 158–172. Lecture Notes in Computer Science, June 2006.

[7] J. Warmer. Octel, a template language for generating structures instead of textstreams. In *Proceedings of the First European Workshop on Composition of Model Transformations*, pages 47–50, July 2006.

[1]It can be downloaded at http://gts.inf.um.es/downloads