Noname manuscript No.
(will be inserted by the editor)

Modularization of model transformations through a phasing mechanism

Jesús Sánchez Cuadrado[1], **Jesús García Molina**[2]

[1] University of Murcia, Spain,
e-mail: jesusc@um.es
[2] University of Murcia, Spain,
e-mail: jmolina@um.es

April 2008

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 mentioned in some works as a rule scheduling or organization mechanism, but without any detail.

In this paper, we present a *phasing mechanism* in the context of rule-based transformation languages. We explain the structure and the behaviour of the mechanism, and how it can be integrated in a language. We also analyze how the mechanism promotes modularity, internal transformation composition and helps to solve usual transformation problems. Besides, we show several examples of application to illustrate the usefulness of the mechanism.

1 Introduction

In recent years a considerable number of model transformation languages has 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 about 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.

As noted in [11], modularization is an important aspect of transformation definitions. Just like programs written in high-level programming languages, a modular transformation definition provides reusability and maintainability. Therefore, a model transformation language should support a "module" construct for the developer to be able to organize a transformation definition as "a coherent and simple structure made of autonomous elements" [14].

In this paper we present a phasing mechanism intended to organize a model transformation definition in several stages or phases, in the context of rule-based transformation languages. The concept of *phase* is provided as a modular construct to deal with the complexity inherent in model transformation. The design of this mechanism is the result of our experimental work writing model transformations with the RubyTL transformation language [18].

The paper is organized as follows. The next section explains different aspects of model transformation that can be addressed with phasing. Section 3 describes the proposed phasing mechanism, while Section 4 discusses how the mechanism makes it possible to address the aspects explained in Section 2. Section 5 shows how and when to use phasing, through some examples. Section 6 presents the related work, and finally Section 7 presents the conclusions.

2 Modularity and model transformation mechanics

Writing model transformations is a complex task, because issues such as establishing the mapping between source and target elements, finding the appropriate rules to express such mappings or dealing with scheduling issues requires a considerable intellectual effort. These difficulties are even increased when achieving transformation quality is a goal.

Winjgaarden and Visser [21] have identified three fundamental aspects of transformation mechanics, which can be used to evaluate transformation approaches. These aspects are: scope, staging and direction. Modularity is recognized as a key aspect to achieve reusable and adaptable transformation definitions [11].

Internal transformation composition [9] is an issue which is closely related to modularity. Composition of

Send offprint requests to:

transformation definitions requires a proper modular construct, providing a composition operator, such that separate transformation definitions can be created as composable units.

In this paper we analyze the proposed phasing mechanism from the points of view of *modularity*, *internal transformation composition*, *staging* and *scope*. Our aim is to show how phases address these aspects, in order to tackle the complexity of writing transformation definitions, as well as improving the quality of such transformation definitions.

2.1 Modularity

Maintainability (i.e. adaptability and the ease of correction of errors) and reusability are recognized as two of the principal software quality factors. They are promoted when software is organized as modular structures [14].

In the context of model transformations, modularity is also a key factor in developing reusable and maintainable transformation definitions. For this reason, transformation languages should provide some "module" construct to allow transformation definitions to be organized in a modular way. Through a modular construct, modular decomposition and composition are possible. Transformation reusability is favoured if a transformation unit has a *specification*, so that the developer only needs to know *what* is transformed into *what*, but not *how* the transformation is done.

In most transformation languages, the basic modular construct is the transformation rule. In [11] a discussion about modularization in current rule-based transformation languages is presented. This discussion concludes that rule integration mechanisms (i.e. the mechanisms to compose rules as modular units) are not enough to satisfy quality requirements for transformation definitions. The fact that the compositional operators are defined at the rule level, causes coupling between rules and reduces reusability and adaptability of rules.

Therefore, as a modular unit, rules are too fine grained to achieve reusability and adaptability. The mechanism we have devised proposes the concept of *phase* as a coarser grained modular unit. It allows decomposition of transformation definitions into smaller and less complex modular units, while allowing composition of existing phases to favor reusability and adaptability.

2.2 Internal transformation composition

As noted in [9], there are two forms of transformation composition: internal and external. The phasing mechanism allows us to tackle *internal transformation composition*, that is, the ability of a tool to compose transformation definitions written in the same transformation language, in order to obtain the required result by combining the results of separate transformation definitions. Similarly to a program written in a general purpose language where program modules are combined, in our approach, transformation modules are combined to form a complete, meaningful transformation definition.

In contrast, *external transformation composition* is the ability to compose transformation definitions written in different transformation languages. This requires tackling interoperatibily between different tools and languages (which may belong to different paradigms), while internal transformation composition only deals with defining the composition mechanism for a single language.

Transformation composition is necessary to allow practical reuse of existing transformation definitions. As we will see, our phasing mechanism allows transformation definitions to be composed in the same way as phases.

2.3 Scope and staging

The three aspects identified in [21] for program transformation tools are *scope*, *direction* and *staging*. Although they are identified for tools that manipulate source programs, not models as understood in MDD, they are also applicable to model transformation languages. Next, we briefly describe these aspects, focusing on their applicability to model transformation.

Wijngaarden and Visser describe **scope** as the area of a model (either source or target) covered by a single transformation step, where a transformation step is usually a single rule application. The *pivot* of a transformation step is defined as the main source element from which a rule resolves. Four main types of transformation steps can be identified between a piece of source model and a piece of target model.

1. *Local source to local target scope transformation step.* A source element (the pivot) can be directly translated to a target element. All the information needed to create the target element is accessible (i.e. can be easily reached) from the source element. Usually, model transformation languages provide good support for this kind of transformation.
2. *Local source to global target scope transformation step.* A source element is transformed into several target elements. Usually, one of these target elements is part of the piece of target model being generated by the rule, while the other target elements need to be allocated in a different part of the target model. These target elements are referred to as *non-local results*. Figure 1(a) shows a graphical example of a local-to-global transformation between a piece of source model and a piece of target model. The elements `a`, `b` and `c` are the pivots of the transformation. They are easily transformed into the elements `x`, `y`, and `w` respectively, and the relationships between them are set. However, the element `z` is also transformed from

the element `a`, but needs to be referenced by the element `w`, which is outside the scope of the rule that creates `x` and `z` from `a`.

3. *Global source to local target scope transformation step.* This situation appears when additional information is needed to create a target element from a source element. This additional information is not easily accessible from the source element being transformed (i.e. the pivot), but a complex query is needed. Figure 1(b) shows a graphical example of a global-to-local transformation between an element of type `B` and another of type `X`. Elements `a1` and `a2` are external information that need to be queried in order to complete the generation of the element `x` from `b`. As can be seen, both `a1` and `a2` are outside the scope of the source element `b`, so this cannot be easily reached from the rule that creates `x` from `b`.
4. *Global source to global target scope transformation step.* This is a combination of the previous two situations. In this paper, we are not specifically concerned about this issue because it can be addressed using the techniques we will explain for the two kinds of transformation steps explained above.

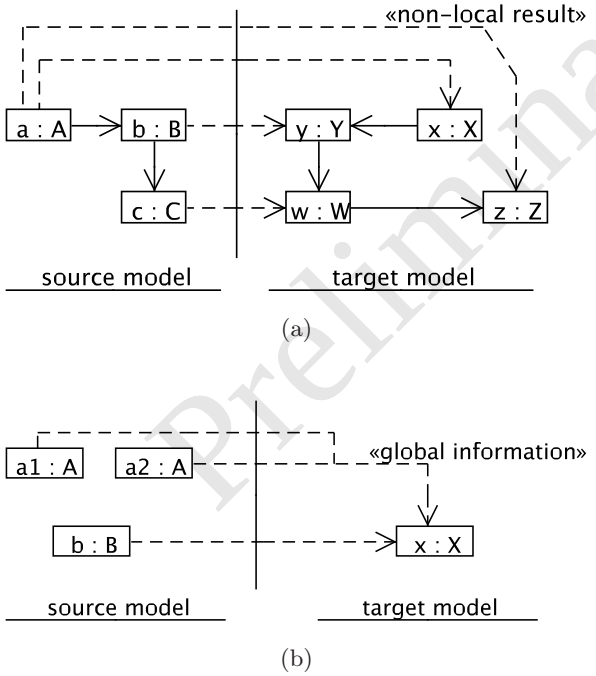

Figure 1 (a) Graphical representation of a local-to-global scope transformation (b) Graphical representation of a global-to-local scope transformation. The solid lines represent references between objects, while dashed ones represent transformation.

For the sake of brevity, in the rest of the paper, we will refer to "global source to local target scope transformation step" as "global-to-local transformation", and "local source to global target scope transformation step" as "local-to-global transformation".

Staging is the ability of a transformation tool to split a transformation definition into several stages. Usually, model transformation languages are *single-stage* tools, that is, the transformation execution consists of applying the transformation rules as a whole. In contrast, a *multi-stage generate* approach allows a transformation definition to be split into several independent stages, each one generating a part of the target model, and then one or more final *merging* stages connect the results of the previous stages.

This aspect is inherent to our phasing mechanism. Each phase is a stage of a complete transformation definition, while merging the results of independent stages can be achieved using the *refinement* feature that will be explained in the following sections.

Finally, **direction** refers to whether a transformation is controlled by the structure of the source model (source-driven) or by the structure of the target model (target-driven). Since this is a characteristic of the underlying transformation language, we do not address this issue in this paper. Instead, the mechanism we propose is general enough to be used with both kinds of languages.

3 Phasing mechanism

In this section we will explain the mechanism to organize a transformation definition in several phases. Both structural and behavioural aspects of the phasing mechanism will be described in detail. In this explanation we will not focus on any concrete transformation language, but we will use a simple textual notation to illustrate the examples. However, the examples proposed in Section 5 will be written in RubyTL [18], using our implementation of the mechanism in this language.

3.1 Mechanism Structure

With a phasing mechanism, a transformation definition is organized as an ordered set of phases. A phase can be either *composite* or *primitive*. A composite phase consists of an ordered set of nested phases (a transformation definition is a composite phase, and it can be seen as a *root phase*). On the other hand, a primitive phase consists of a set of rules.

According to this definition, phases are organized hierarchically, in a tree-like structure. When a transformation definition (or root phase) is executed, all its nested phases are executed, in a depth-first fashion. In this way, a composite phase is executed by executing all its nested phases, while a primitive phase is executed by evaluating its rules (i.e. the execution semantics of a primitive phase is similar to a plain transformation definition).

The phasing mechanism structure can be defined by an abstract syntax, as is shown in Figure 2, where the

concepts involved and their relationships are made explicit. It is worth noting that a transformation definition (metaclass `Transformation`) is considered a specialization of `CompositePhase`, so it is always composed of one or more phases (either composite or primitive). Since a transformation definition is a kind of phase, the transformation engine can handle transformation definitions as phases. In Section 4.2 we will explain how this feature promotes reusability through internal transformation composition.

There are two ways of executing phases, *explicitly* and *implicitly*. In the former case the user is in charge of setting the phase execution order, while in the latter the transformation engine execute phases in the same order as they appear in the transformation definition. This issue will be treated in depth in Section 3.2

A phase has zero or more by-value *parameters*, which need to be set when the phase is going to be executed. A phase parameter has an optional default value, that will be used if no other value is provided when the phase is launched. If a phase has one or more parameters, then it is compulsory to launch its execution explicitly, unless all of them have a default value.

Moreover, since a transformation definition is a kind of phase, it allows us to parametrize a concrete transformation execution using the phase parameters. For instance, in a transformation between a UML class model and a Java model, a parameter of the transformation definition could be the qualified name of the Java package where the Java classes should be placed.

Finally, each phase has a conditional expression which must hold true in order for the phase to be executed.

3.1.1 Phase scoping

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.

This property is useful to allow rules with the same source and target pattern to act on the same elements but to behave differently according to the current transformation state, i.e. depending on the phase it belongs to.

3.1.2 Refinement rule

As can be seen in the abstract syntax of Figure 2, 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. This feature is called *transformation refinement* and it is explained further in Section 3.2.

We distinguish between two kind of primitive phases. We will call *generate phases* to those phases which do not have any refinement rules, and *refinement phases* to those phases that have at least one refinement rule.

3.1.3 Trace query

A *trace query* is a function (named `trace_query`) that takes a source element and a target type as arguments, and returns one or more target elements of the specified type. The returned target elements are related by the trace to the input source elements. The implementation of this function, and how a rule can call it, depend on the concrete transformation language. In Section 5 we will show how it is used in our implementation in the RubyTL transformation language.

3.2 Mechanism behaviour

The basic behaviour of the phasing mechanism can be explained without considering a concrete transformation language, but the transformation refinement capability is dependent on the kind of transformation language. In this section, the basic execution mechanism of phases is explained, and then the transformation refinement capability is defined with respect to rule-based transformation languages. First, however, the notion of *transformation state* is introduced.

3.2.1 Transformation state

A transformation is a process in which a target model is generated from a source model, by executing a transformation definition. At each time, this process is characterized by the *transformation state*, defined by the state of all generated target elements. A transformation starts with an initial state (the source model) and goes through a sequence of intermediate states until a final state (the target model) is reached. Usually, a change in the transformation state is provoked by a transformation rule. Figure 3(a) shows a schematic representation of the evolution of the transformation state through n intermediate states, where e_0 denotes the initial state and e_f denotes the final state.

When the phase construct is present in the transformation language, each phase involves a set of intermediate states, so that the number of states is reduced and they have a more abstract meaning. Figure 3(b) shows how the transformation of Figure 3(a) is decomposed in two phases, each involving a set of intermediate states.

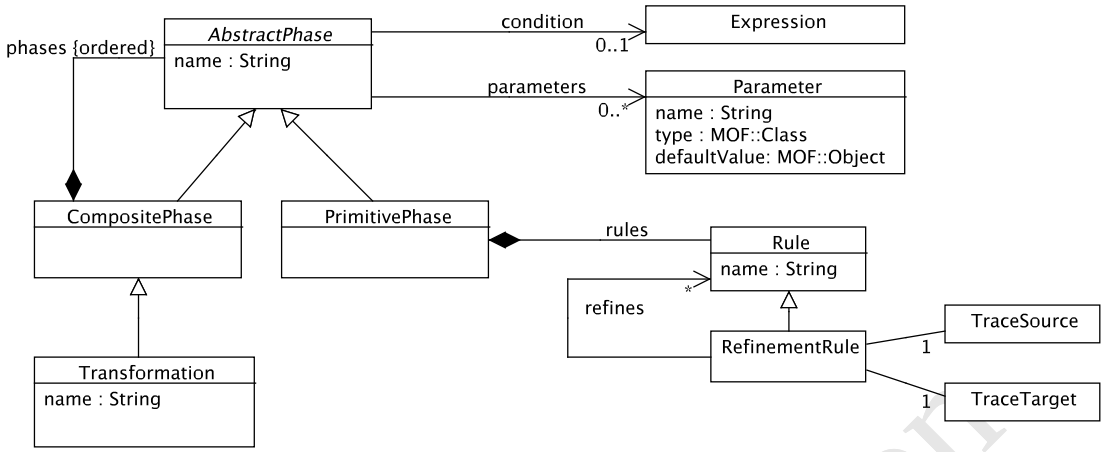

Figure 2 Part of the abstract syntax of a model transformation language providing phases represented as a metamodel. The transformation language considered is a rule-based language, which is also extended with the concept of *refinement rule*

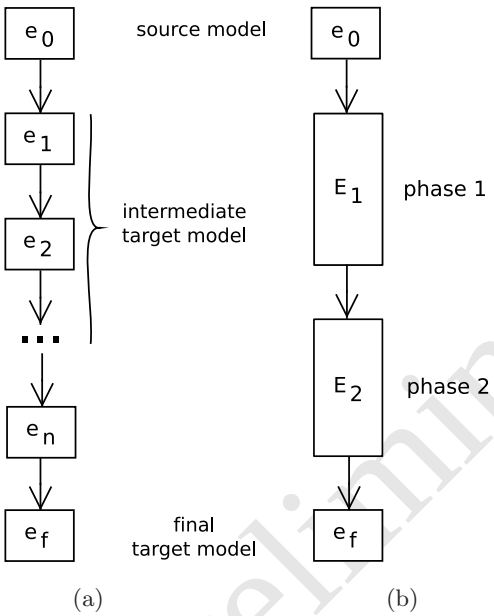

Figure 3 a) Representation of transformation state evolution from the initial state to the final state. (b) Representation of how the transformation state evolves at the same time that phases are executed.

3.2.2 Phase execution

The execution of a phase depends on whether the phase is primitive or composite. In the case of a primitive phase, it 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.

In the case of composite phases, the execution mechanism is divided in two different cases, depending on whether the nested phases are executed implicitly or explicitly. It is important to note that a transformation definition is a composite phase, i.e. the root phase.

Implicit execution. This is the simplest case, where the transformation engine executes each phase in the same order as it appears in its enclosing phase, the phase parameters are set to their default values. Since the default value of a parameter is optional, it may be that a parameter does not have any default value. In this case, the phases need to be executed explicitly because the actual parameters must be set before execution, as explained below.

Explicit execution. In this case, the user is provided with a DSL to schedule the transformation execution by launching phases. Figure 4 shows an excerpt of its abstract syntax. This DSL provides the `PhaseExecution` statement which starts the execution of the named phase. Since a phase has zero or more formal parameters, a phase execution can be provided with the actual parameters, if needed.

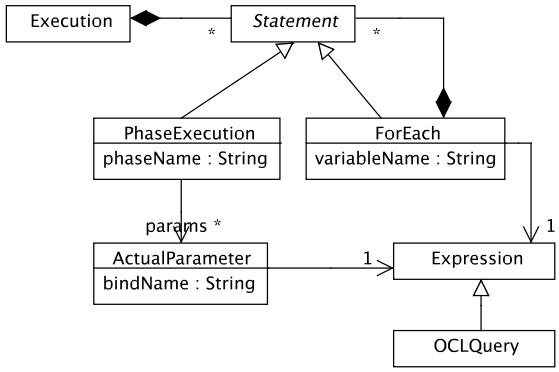

Figure 4 Metamodel that defines the abstract syntax of a simple DSL aimed to set the explicit execution of phases.

The `ForEach` statement is provided to allow the same phase to be executed multiple times. This statement allows a collection to be iterated, setting the iteration variable specified by `variableName` to the corresponding value in each iteration. As an example, one kind

of expression that can be used to get a collection of model elements is an `OCLQuery` (a query in the OCL language). In any case, the expression language of the underlying transformation language can be used to compute expressions in this DSL.
The following example shows the use of the proposed DSL to launch three phases. First, phase `example-a` is launched without actual parameters, then phase `example-b` is launched setting the actual parameters `paramx` and `paramy` to the values of the corresponding expressions, and finally the phase `example-c` is launched ten times according to the OCL expression `Sequence { 1..10 }`, and each time the phase is launched, the parameter `paramz` is set to the iteration variable value.

```
execute phase example-a

execute phase example-b
  where paramx = A.allInstances()->
                   collect(a | a.name)
        paramy = 10

foreach i : Sequence { 1..10 }
  execute phase example-c
    where paramz = i
```

Finally, regarding phase execution, since generate phases do not interact with each other, they can be executed concurrently. The only restriction is that a refinement phase cannot be executed until all previous generate phases have finished (i.e. refinement phases act as a join for the transformation threads).

3.2.3 Conditions for phase execution

There are two conditions that can prevent a phase from being executed. The first occurs when the phase precondition is not satisfied, and the second when the explicit execution mode is used. In both cases, if a composite phase is not executed, neither are any of its nested phases.

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.

When phases are scheduled for execution explicitly, the user may decide not to select for execution one or more phases. This can be useful to give the user control over the transformation process, allowing certain transformation customization to be achieved. However, some care must be taken to avoid breaking a transformation definition, since dependencies between phases may exist.

3.2.4 Transformation refinement

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 explicitly refine the work done in previous phases (i.e. modifying existing target elements). This can be seen as the *composition operator* for phases.

The simplest way to provide a composition operator is to explicitly use the name of the rules whose results will be refined [17], 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. When a phase needs 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 a *refinement rule*, as mentioned above. This kind of rule has a `trace_source` and a `trace_target` pattern which match against the trace, instead of against a source and target model as normal rules do. It is worth 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 has 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.

It is important to note that in our approach trace links are automatically collected in each rule application.

This makes collecting trace links transparent for the transformation developer, as well as preventing explicit tracing code to be scattered through all the transformation rules.

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 5. In the second phase, new target elements `c1` and `c2` of type `C` are created from the `a1` and `a2` 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`. Because it is a refinement rule, no new targets elements are created, but the previously created target elements are refined.

Given the trace of Figure 5, 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 element `b3` is not refined, since `a3` is not related to any element of type `C` by the trace.

```
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
```

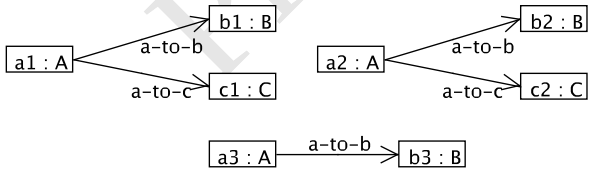

Figure 5 Transformation trace example. Source and target elements are related by an arrow with the name of the rule which performed the mapping.

Finally, the *trace query* function presented in Section 3.1.3 is a complement to the transformation refinement capability. Since this function allows us to retrieve target elements that are related by the trace to a source element, it makes it possible read them, ensuring that they exist.

For instance, in the case of the transformation trace shown in Figure 5, the exhaustive list of results a trace query function will return is the following. The notation { } means a list of elements.

```
trace_query(a1, B) == { b1 }
trace_query(a2, B) == { b2 }
trace_query(a3, B) == { b3 }
trace_query(a1, C) == { c1 }
trace_query(a2, C) == { c2 }
trace_query(a3, C) == {  }
```

In the case of `trace_query(a3, C)` the result is an empty list, since there is no trace relationship between `a3` and an element of type `C`.

4 Aspects addressed by the mechanism

In this section we will discuss how the phasing mechanism addresses the transformation aspects considered in Section 2, namely modularity, internal transformation composition, staging and scope.

4.1 Modularity

A phase is a modular construct that allows the *decomposition* of a transformation into a set of less complex transformations. In this way, instead of writing a transformation definition as a set of rules, all dealing with the whole source and target models, the definition is divided into smaller and more meaningful set of phases. Each phase deals with a part of the source model that it is transformed into a part of the target model.

On the other hand, phases can be composed so that a complex transformation can be built up from smaller transformations. The composition is achieved by means of refinement rules that resume the work done by rules in previous phases, and refine the existing target elements, making it possible to compose them.

In the model transformation context, *maintainability* refers to the ease with which a transformation definition can be modified to correct errors, or to be adapted to deal with changes in the source and target metamodels. Phases allow us to find errors more easily, and to extend a transformation definition without modifying the existing transformation text.

Finding errors is made easy in two ways. Firstly, because rules can be organized in meaningful units, each one dealing with a concern of the transformation definition (as will be illustrated in Section 5.3). When an error affecting a concern arises, it will be located in only one place: the phase which implements the concern. Secondly, since a phase is executable by itself (provided that it does not depends on other phases), it can be tested independently of other phases.

With regard to *extensibility*, once a transformation has been written, in general it is always possible to write a refinement phase that extends it. For instance, given

an existing transformation between a UML class model and a Java model, that creates Java classes from UML classes, a possible extension is to generate JUnit test classes for each existing Java class. However, modifying the original transformation causes two problems: (1) an already existing, probably working, transformation definition need to be modified, and (2) it implies scattering the logic for generating test cases and test methods through all the definition. In addition, this solution would make the transformation non-reusable in contexts where JUnit test cases were not useful. The solution using phases is to extend the UML to Java transformation by means of an independent refinement phase, whose work is based on the results produced by the original transformation. From each existing Java class created from a UML class, a new JUnit test case is created. In this way, the UML to Java transformation remains independent of the extension.

Nevertheless, there are two contraints with regard to when it is possible to extend an existing transformation definition. First of all, since refinement rules work on existing mappings, the extended transformation definition must provide at least one mapping which makes sense to be refined by the new transformation definition. Secondly, this approach for extensibility is constructive, in the sense that a new transformation cannot prevent the extended transformation from creating model elements as defined in its rules.

In addition to extensibility, *reusability* is also achieved, since a transformation definition can be written as a reusable component. For instance, in a transformation from a UML class model to a certain web architecture, the transformation rules to create business objects can be encapsulated in a separate transformation definition, intended to be reused to generate other architectures.

It is important to notice that the reusability of a transformation definition is limited by its source and target metamodels. Even though the functionality of a transformation definition T1 can be reused within another transformation definition T2, if the source and target metamodels used in T1 are not "compatible" with the metamodels used in T2, then it is not possible to reuse T1 directly.

In addition, phase parameters allow us to parametrize a transformation with external information, so that it is possible to make the transformation definition more general, thus improving its reusability.

To achieve the goals of reusability and extensibility, a *specification* of what a transformation provides is needed. Otherwise, all the transformation rules must be examined to determine the relationships they establish between the source and the target model.

From the point of view of the phasing mechanism, the set of trace relationships established by a phase is its interface with respect to other phases. The specification of a transformation definition (or a phase) is defined by its interface. Thus, from a transformation definition, a tool can extract the specification in the form of the trace elements that will be generated (i.e. pairs of source and target metaclass).

For instance, for the UML to Java transformation mentioned above, the specification extracted with a documentation tool would be the following.

```
uml2java:
 - parameters
   * java_package : String
   * attr_visibility : Java::VisibilityKind

 - trace relationships
   * UML::Class -> Java::Class
     A UML class is mapped to a Java class

   * UML::Attribute -> Java::Field
     Each attribute is mapped to a Java field.
     The visibility is set to the
     attr_visibility parameter.

   * UML::Attribute -> Java::Method
     Each attribute is mapped to a public
     get method.
     Each attribute is mapped to a public
     set method.

   * UML::Operation -> Java::Method
     Each operation of a UML class is mapped to
     a Java method.
```

From this specification, it is possible to know that this transformation maps each UML class to a Java class, and for each UML attribute a field and a pair of accessor methods are created. Additionally, operations are mapped to methods.

Also, it is possible for a transformation compiler, to ensure that certain trace relationships exist when defining a refinement rule.

4.2 Internal transformation composition

Internal transformation composition is another aspect of model transformation that can be achieved with this mechanism.

The integration of existing transformation definitions within another transformation definition is transparent, since the transformation engine treats existing transformation definitions as if they were phases written within the current transformation definition. When an existing transformation is executed as a phase, the transformation engine executes the transformation on the current transformation state, allowing the transformation to refine the partially created target model.

Since each transformation definition is usually written in a different file, we need a way of *importing* them into a given transformation definition in order to reuse

such existing definitions. In Section 3.2.2 a DSL to execute transformation definitions was presented. To give support to the composition of an existing transformation, this DSL includes a *transformation import* statement that provides the ability of reading an existing transformation definition and managing it as a phase.

The approach we propose to reuse existing transformations is to define an "integrating transformation definition", that is, a definition which imports one or more existing transformation definitions and defines one or more refinement phases to merge and adapt the results of the imported transformations to a specific context, using a combination of refinement rules and *trace_query* function calls. The following code is an example that imports two transformations `T1` and `T2`, and merges the results of both of them using the refinement phase `mergeT1_T2`.

```
phase mergeT1_T2
  refinement_rule merge
  ...

scheduling
  import transformation T1
  import transformation T2
  execute phase mergeT1_T2
```

Using the `scheduling` construct, the phases are explicitly scheduled to be executed. In this case, since we want to execute existing transformation definitions, the `import transformation` construct is used to specify that both transformation definitions `T1` and `T2` must be imported within the current transformation definition, and be executed as phases. Finally, the refinement phase `mergeT1_T2` is executed.

4.3 Staging

As we explained in Section 2.3, *staging* is an aspect of transformation languages that states whether is it possible to implement a transformation definition in several stages or steps. Each stage can perform a different traversal to the source model, and creates a certain part of the target model.

The proposed phasing mechanism allows **multi-stage generate** transformations to be implemented, that is, every stage generates a piece of the target model that is eventually merged with all other pieces to form the final target. The ability of our proposal to implement multi-stage generate transformations is inherent to the mechanism, since each phase is a stage of the transformation, and the same phase can be executed multiple times, as explained in Section 3.2.

Usually, one or more generate phases are in charge of creating some part of the target model. They are executed sequentially without dependencies between them. Afterwards, one or more refinement phases merge the results produced by previous phases.

4.4 Local-to-global scope

Local-to-global scope transformations imply generating several target elements from the same source element.

As we explained in Section 2.3, usually one of these target elements is part of the piece of target model being generated by the rule, but the other target elements need to be allocated in different parts of the model (i.e. the non-local elements), which may not have already been generated.

As noted in [21], local-to-global scope transformations where the target elements are related (as is the case of Figure 1(a)) can be implemented using multi-stage generate transformations. Since our phasing mechanism supports this kind of transformations, a local-to-global transformation is implemented as several generate phases, each one implementing direct transformations, followed by one or more refinement phases that allocate the target elements in the correct place (i.e. set the references to the proper elements).

For instance, the transformation problem shown in Figure 1(a) can be solved in three phases. Figure 6 shows how the target model is incrementally built in each phase. The first phase is a generate phase that builds the target model by means of simple one-to-one rules that create the objects `x`, `y` and `w`, and set the relationships between them. In the second phase, the non-local element `z` is created from the element `a` by means of another one-to-one rule. The third phase is a refinement phase that is in charge of connecting the element `z` to the element `w`. Since both `w` and `z` already exist, a combination of a refinement rule and a *trace_query* can be used to get both elements, and connect them.

In Section 5.4 we will show a concrete example of local-to-global transformation, where a DSL is compiled to another DSL of a lower abstraction level.

4.5 Global-to-local scope

In *global-to-local* scope transformations, the generation of a single target element from the source element (i.e. the pivot) requires additional information, that cannot be directly reached from the pivot, but a more complex query is needed. This additional information can be either external information (i.e. configuration information), context information, or information that belongs to a different part of the source or target model.

Usually, the solution is to place the query in helper functions or patterns that perform the navigation and retrieve the proper elements. However, as noted in [2], sometimes this solution couples the helper and the rules that use them, making the helpers and the rules non-reusable.

Phasing mechanism features, such as iteration and phase parameters, provide a means to decouple helper functions that perform navigation from transformation

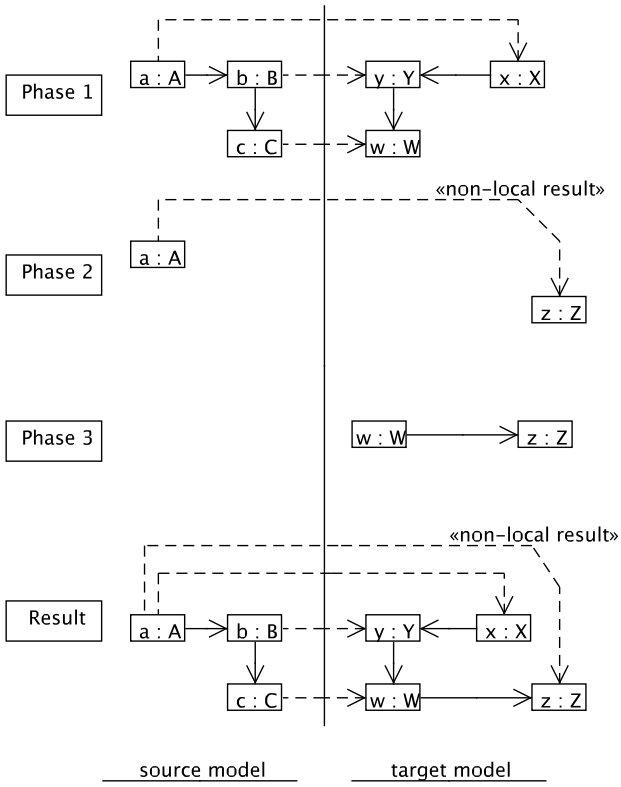

Figure 6 A graphical representation of how to solve a local-to-global transformation in three phases. The first two phases are generate phases, while the third one is a refinement phase. The solid lines represent relationships between objects, while dashed ones represent transformations. Each graphic shows the elements involved in the transformation. The last graphic shows the final result.

rules. This is convenient in cases where either the navigation logic or the transformation rules can be reused in other contexts.

In the following example, a phase is parametrized to accept the result of a query. The query is executed before executing the transformation, and the result is used as the actual parameter of the phase. The query `SomeQuery` collects elements needed by the rules of the phase to perform their work. These elements are the global information.

```
phase example
  parameter query_result : Collection<X>

  rule example
    use query_result

scheduling
  execute phase example
    where query_result = SomeQuery
```

In Section 5.5 we will show an example that separates navigation logic from transformation rules, making both of them reusable.

5 Usefulness of the approach

In this section we will show several examples to illustrate how the phasing mechanism can be useful to solve model transformation problems. In the examples, our implementation of the mechanism in the RubyTL transformation language [18] will be used.

RubyTL is a hybrid rule-based transformation language, that provides an imperative part based on the Ruby language, and a declarative part based on rules and bindings. A binding is a special kind of assignment that specifies "what is transformed into what", letting the transformation algorithm resolve the rule able to transform the right part of assignment into the left part [8]. Instead of introducing RubyTL completely here, we will comment those parts of the examples that may be not clear.

5.1 Tackling complex transformation

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

In this way, the developer can map his or her mental scheme directly onto 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 a sequence of steps, where steps creating target elements coexist with steps merging these target elements. This decomposition corresponds to the first level of modular decomposition shown in Figure 7. The empty circles represent generate phases, while the grey circle represents a refinement phase that merges the results of the previous phases.

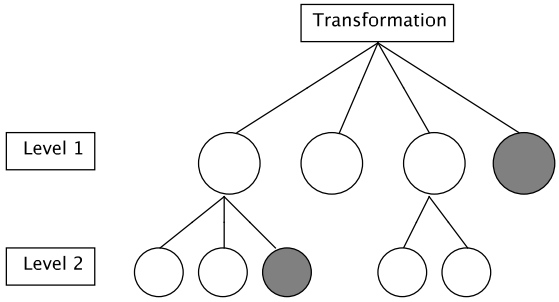

Figure 7 Graphical representation of the decomposition of a transformation in phases. Two levels of decomposition are considered. Empty circles represent generate phases, while grey circles represent refinement phases.

Depending on the complexity of the resulting phases of the first level, they can be further decomposed. The decomposition at the second level is usually caused by problems inherent to the mapping between the source

and the target metamodel. For instance, if a local-to-global transformation needs to be solved, the phase can be split into several nested phases. This kind of decomposition corresponds to the second level of modular decomposition shown in Figure 7.

The decomposition in several levels is possible because the structure of the phasing mechanism allows nested phases to be defined. Even though the decomposition can be done at any nesting level, usually no more than two levels make sense. In general, the decomposition at the first level is intended to make the transformation definition more readable, maintainable and reusable. Once this quality aspects has been addressed, the second level of decomposition is intended to solve a specific transformation problem.

It is worth noting that the decomposition at the first level usually establishes a "main modularization", which may impose restrictions on how to deal with new requirements. Since a transformation definition can be modularized in only one way at a time, the mechanism suffers from the problem of the *dominant decomposition* [20] (i.e. such concerns that do not align with the chosen modularization may end up scattered and tangled). This means that some care must be taken in the design of the phase decomposition, since it will affect the way new requirements can be accommodated. In Section 5.3 an example of phase decomposition that promotes quality properties such as evolvability and reusability is presented.

We recall that the composition operator for phases is the refinement rule (i.e. a phase with at least a refinement rule). In Figure 7, this is represented by a grey circle, which represents a refinement phase. Usually the refinement phases are executed after all generate phases, thus composing their results.

5.2 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) [8], since the internal scheduling mechanism of the language cannot ensure that the target elements are available when they are going to be read [2]. 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, generate phases deal with creating some part of the target model, that is, adding elements to the transformation state. Thus, when a refinement phase is executed, it can safely refer to a previous intermediate transformation state (i.e. the part of the target model that has already been generated). This means that it is safe to read the target model and 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 a refinement rule always matches to both existing source and target elements. In fact, a tool can analize the transformation phases and determine whether the queries are consistent.

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, and if the source model has circular dependences between classes, it may be that primary keys are not available when needed. An example addressing this problem can be found in [17], where a transformation definition is organized in three phases to solve the mentioned dependencies: in the first phase tables and columns are created from classes and attributes respectively, then primary keys are set in the second phase, and finally, the third phase computes the foreign key columns based on the primary keys set on the previous phase.

In the example of Section 5.4, this feature will be used to address local-to-global transformations.

5.3 Improving modularity

To illustrate how the phasing mechanism can improve modularity of transformations, we will use the example of the *examination assistant* explained in [11] where problems related to scattering and tangling are identified in a transformation between a model representing exams and questions, and a user interface model based on the *Model-View-Controller* (MVC) pattern.

Tangling is an anomaly that appears when a single module contains pieces of functionality which belong to different concerns. On the other hand, scattering is spreading a single piece of functionality across several modules [6][11].

Figure 8 shows the metamodel that describes exams and exam questions. An exam is composed of one or more exam questions. There are two concrete kinds of exam questions: open and multiple choice. Figure 8 shows the class hierarchy: an abstract class `ExamElement` and two subclasses, `OpenElement` and `MultipleChoiceElement`.

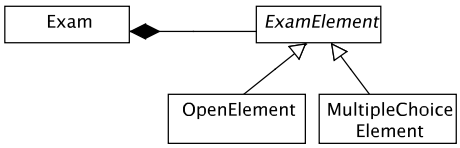

Figure 8 Metamodel for the description of exams and exam questions.

The target model represents a MVC model conforming to the metamodel shown in Figure 9. There are three abstract classes: `ExamItem`, `View`, `Controller`, corresponding to model, view and controller classes respectively. Each one has concrete subclasses that represent specializations for open and multiple choice questions.

The *examination assistant* system supports teachers in performing computer-based examinations. A teacher will create a source model describing an exam, that will be loaded in the system. When a student wants to take the exam, a transformation execution creates a set of objects whose execution allows the student to complete the exam. Depending on the implementation of the concrete model, view and controller classes, the system will behave differently. For instance, the concrete controller classes shown in Figure 9 could implement a *self-test* mode, where students can check their answers. A possible evolution of the system could be creating new concrete controller classes, for instance to implement a *tutorial* mode, where students cannot enter information, but just see the answers.

Organizing this transformation definition in a single set of rules causes tangling and scattering problems. The transformation definition shown below is organized in two rules. Each one transforms a concrete kind of exam question to the corresponding concrete model, view and controller elements.

```
rule 'openQuestion'
  from Exam::OpenElement
  to   MVC::Open,
       MVC::OpenView,
       MVC::OpenController
  mapping do |element, model, view, controller|
    model.observer  = view
    view.controller = controller
    view.fontName   = 'Times'
  end
end

rule 'multipleChoiceQuestion'
  from Exam::MultipleChoiceElement
  to   MVC::MultipleChoice,
       MVC::MultipleChoiceView,
       MVC::MultipleChoiceController
  mapping do |element, model, view, controller|
    model.observer  = view
    view.controller = controller
    view.fontName   = 'Times'
  end
end
```

In this transformation definition, tangling appears because each rule refers to several concerns in the target pattern: the model, the view and the controller. Moreover, the code to set view properties (e.g. `view.fontName = 'Times'`) is scattered through the two rules. Tangling and scattering limit evolution of the transformation definition, since a change in one concern implies modifying other concerns. Reusability is also compromised since each concern is not specified separately. Therefore, only the transformation as a whole can be reused.

To solve these problems, the transformation definition is rewritten in five phases, using the refinement mechanism explained in Section 3.

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

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

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

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

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

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

phase 'mvc' do
  refinement_rule do
    from Exam::ExamElement
    to   MVC::ExamItem, MVC::View,
         MVC::Controller
    mapping do |element, model, view, controller|
      model.observer  = view
      view.controller = controller
    end
  end
end
```

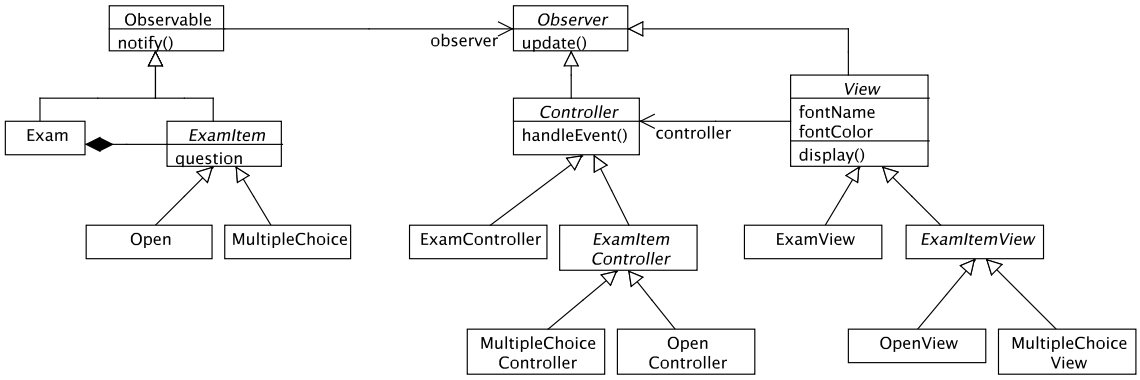

Figure 9 Metamodel for the description of the MVC pattern.

```
phase 'layout' do
  refinement_rule do
    from Exam::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, a refinement rule 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, `ExamItem`, `View`, `Controller` are also supertypes of the target metaclasses of the refined rules (actually all rules of the three first phases).

In the case of *scattering*, it is solved by the phase `layout`, which has a refinement rule intended to set view properties. Thus, the rules `OpenQuestionView` and `MultipleChoiceView` are refined, so that layout assignments are not scattered through view rules, but they are placed in this final phase.

This organization in phases promotes modularity. On the one hand, it leads to an improvement in the evolvability of the transformation definition. For instance, if the *tutorial* mode is introduced, the concrete controller classes specified in the transformation definition need to be replaced by other ones. Since tangling has been solved, only the `controller` phase needs to be changed to replace old controller classes with the new ones. In the case of scattering, since the layout style is set only in one place (i.e. the `layout` phase) then it is possible to change it without affecting other parts of the transformation.

On the other hand, using the internal transformation composition mechanism explained in Section 4.2, it is possible to separate this transformation into reusable transformation components, that can be composed to produce different transformation definitions. The following code shows an "integrating transformation definition" which composes several, separate transformation definitions.

```
scheduling do
  import_transformation 'model'
  import_transformation 'view'
  import_transformation 'tutorial_controller'
  import_transformation 'mvc'
  import_transformation 'yellow_layout'
end
```

As can be seen, we can easily replace the `controller` and `layout` phases by another ones. Actually, it is possible to have several integrating transformation definitions like this, so that we can provide different transformation solutions just by composing existing reusable transformation definitions.

As we said, the *examination assistant* example is used in [11] to analyze modularization issues in rule-based transformation languages. The solution proposed in this section shows that phases are a proper construct to solve such issues.

5.4 DSL compilation : local-to-global transformations

Compilation of DSLs is a kind of transformation definition where local-to-global transformations appear frequently. A concept in the DSL is translated to several elements of the low-level language, and these elements need to be allocated in different places of the target model.

In this section we will show an example based on translating a simple DSL for specifying preference options of an application to another DSL that specifies widgets, and how they are laid out in a dialog box.

Figure 10(a) shows the metamodel for the DSL intended to define preferences. Preference groups can be

defined, and each group has one or more options. Figure 10(b) shows the metamodel for the DSL intended to specify a dialog box, including which widgets are part of dialog and how they are laid out. To define a dialog box, the widgets that compose the dialog are specified (`definitions` relationship), and a main layout needs to be chosen (relationship `layout`). There are three concrete layout classes: `VFlowLayout` represents a layout where elements are placed vertically in a row, `HFlowFlayout` represents a layout where elements are placed horizontally in a row, and `WidgetUse` is a reference to a widget, which will be placed according to the owning layout.

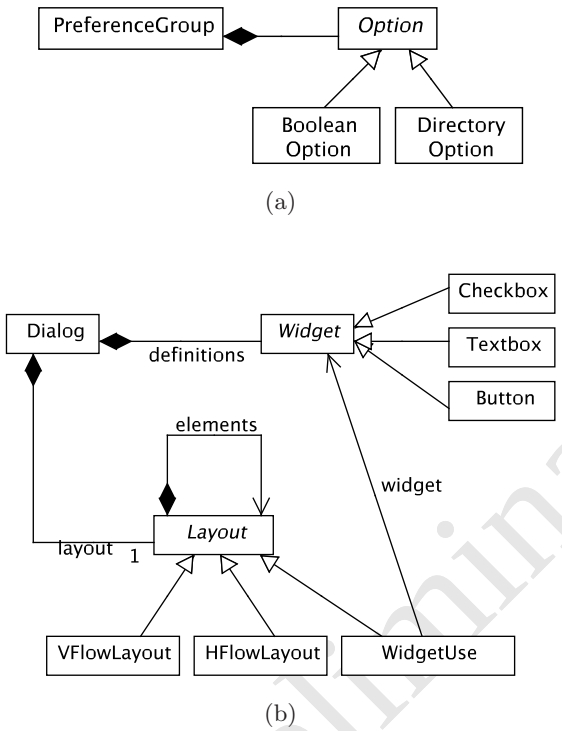

Figure 10 (a) Abstract syntax of a simple DSL intended to specify preference options of an application. (b) Abstract syntax of a DSL intended to specify the layout of user interface widgets.

The following piece of code represents a model conforming to the preferences DSL metamodel, using a possible textual concrete syntax.

```
preference-page 'Path options' {
  boolean option 'Share files'
  directory option 'Incoming dir.'
}
```

This model will be translated to the following model (again expressed using a possible textual concrete syntax), which conforms to the dialog specification DSL metamodel.

```
dialog 'Path options' {
  checkbox  shareFiles 'Share files'
  textfield incomingDirText 'Incoming dir.'
  button    incomingDirButton '...'

  layout vertical {
    shareFiles
    layout horizontal {
      incomingDirText
      incomingDirButton
    }
  }
}
```

There are three important points to be taken into account in this translation:

- Each preference page is translated into a dialog box, and a vertical layout, which is the main layout of the dialog.
- A boolean option is translated into a checkbox widget definition, while a directory option is translated into two widget definitions: a text field and a button. In addition to these two widgets, an horizontal layout is created to align them horizontally.
- The main vertical layout contains a reference to the `shareFiles` widget (represented by `WidgetUse` in the abstract syntax). The horizontal layout contains a reference to the `incomingDirText` and `incomingDirButton` widgets, and make them horizontally aligned.

Figure 11 shows these mappings graphically. As can be seen, the relationships are not trivial. Each preference option is transformed to more than one target element, which need to be allocated in different parts of the target model. These mappings are examples of *local-to-global scope transformations*. For instance, from every `DirectoryOption` the following target elements are created: a `TextBox` and a `Button`, a `HFlowLayout`, and two `WidgetUse` (one for each widget). These target elements are related, and their relationships need to be set in the transformation definition. The two `WidgetUse` objects are connected to the `HFlowLayout` object, and they have a reference to the `TextBox` and `Button` widgets.

As can be seen, the target elements interact among them (i.e. they have relationships that need to be set), and the integration process cannot be done in a single step, thus we will use a multi-stage generate transformation: several generate phases performing direct mappings are implemented, followed by one or more refinement phases that merge the results.

In this case, we can organize the transformation definition into three phases, as shown below. The first phase creates the dialog, and the corresponding widget definitions. The *binding* `dialog.definitions = group.options` implicitly makes the result of the `boolean` and `directory` rules to be allocated in the proper place (i.e. objects of type `Checkbox`, `TextBox`, `Button` are assigned to the `dialog.definitions` reference). The second phase is in

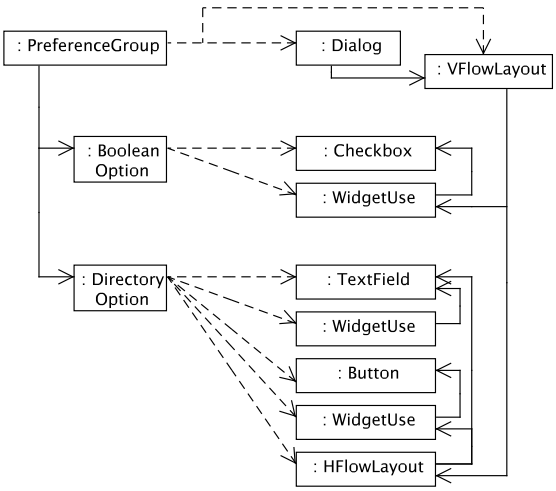

Figure 11 Graphical representation of the transformation between a source model and a target model representing preference options and widgets respectively. Solid lines represent references between elements, while dashed lines represent transformations.

charge of the layout. As explained above, for each widget the corresponding layout elements are created. Finally, the third phase is in charge of the integration. The refinement rule `widget_use` connects objects of type `WidgetUse` to objects of type `Widget`. In the same way, the refinement rule `dialog` connects the main layout to the dialog created in the first phase.

```
phase 'widget_definition' do
  rule 'group' do
    from DSL::PreferenceGroup
    to   Widgets::Dialog
    mapping do |group, dialog|
      dialog.definitions = group.options
    end
  end

  rule 'boolean' do
    from DSL::BooleanOption
    to   Widgets::Checkbox
  end

  rule 'directory' do
    from DSL::DirectoryOption
    to   Widgets::TextBox, Widgets::Button
  end
end

phase 'layouts' do
  rule 'group' do
    from DSL::PreferenceGroup
    to   Widgets::VFlowLayout
    mapping do |group, layout|
      layout.elements = group.options
    end
  end

  rule 'boolean' do
    from DSL::BooleanOption
    to   Widgets::WidgetUse
  end

  rule 'directory' do
    from DSL::DirectoryOption
    to   Widgets::HFlowLayout,
         Widgets::WidgetUse,
         Widgets::WidgetUse
    mapping do |option, layout, text, button|
      layout.elements = [text, button]
    end
  end
end

phase 'integration' do
  refinement_rule 'widget_use' do
    from DSL::Option
    to Widgets::Widget, Widgets::WidgetUse
    mapping do |option, widget, widget_use|
      widget_use.widget = widget
    end
  end

  refinement_rule 'dialog' do
    from DSL::PreferenceGroup
    to   Widgets::Dialog, Widgets::Layout
    mapping do |group, dialog, layout|
      dialog.layout = layout
    end
  end
end
```

Another implementation option is to use the *trace_query* concept introduced in Section 3. It is possible to rewrite the second phase as shown below, so that the third phase is no longer needed. Now, the rules `group`, `boolean` and `directory` use the `trace_query` function to locate existing target elements (i.e. created in previous phases) that are related to the target elements they are creating. For instance, the rule `group` uses `trace_query(group, DSL::Dialog)` to get the dialog where the layout needs to be connected. The first argument is a source element, while the second argument specifies the type of the target elements that are related by the trace to such a source element.

```
phase 'layouts' do
  rule 'group' do
    from DSL::PreferenceGroup
    to   Widgets::VFlowLayout
    mapping do |group, layout|
      layout.elements = group.options

      dialog = trace_query(group, DSL::Dialog)
      dialog.layout = layout
```

```
    end
  end

  rule 'boolean' do
    from DSL::BooleanOption
    to   Widgets::WidgetUse
    mapping do |option, widget_use|
      widget_use.widget = trace_query(option,
                              Widgets::Checkbox)
    end
  end

  rule 'directory' do
    from DSL::DirectoryOption
    to   Widgets::HFlowLayout,
         Widgets::WidgetUse
         Widgets::WidgetUse
    mapping do |option, layout, text, button|
      text.widget   = trace_query(option,
                              Widgets::TextBox)
      button.widget = trace_query(option,
                              Widgets::Button)
      layout.elements = [text, button]
    end
  end
end
```

In this example, we have shown two alternatives to implement local-to-global transformations: using refinement rules and *trace_query* function calls. On the one hand, with the *trace_query* approach it is possible to make each rule responsible of allocating the target elements it is creating, which makes the transformation definition less verbose and easier to understand. On the other hand, using refinement rules there is less coupling between phases, since the piece of code to connect the target elements is not scattered through several phases, but it is placed in a final refinement phase.

5.5 State Machine configurations: global-to-local transformations

In this section global-to-local transformations are illustrated by means of a transformation example that deals with the configurations of a UML-like state machine. The example will show how phase features, such as parameters and phase iteration, allow a phase to be provided with the external information needed to perform its tasks, so decoupling transformation rules from queries that get the external information.

A state machine configuration is defined as the set of active states at a given moment. In this way, the transformation presented below computes all configurations of a state machine, and transforms each configuration into a graphic, where non-active states are drawn as empty circles, active states as filled circles, composite states as boxes, and transitions as lines. Figure 12(a) shows a simplified version of the UML state machine metamodel, while Figure 12(b) shows a simple metamodel to represent graphical figures.

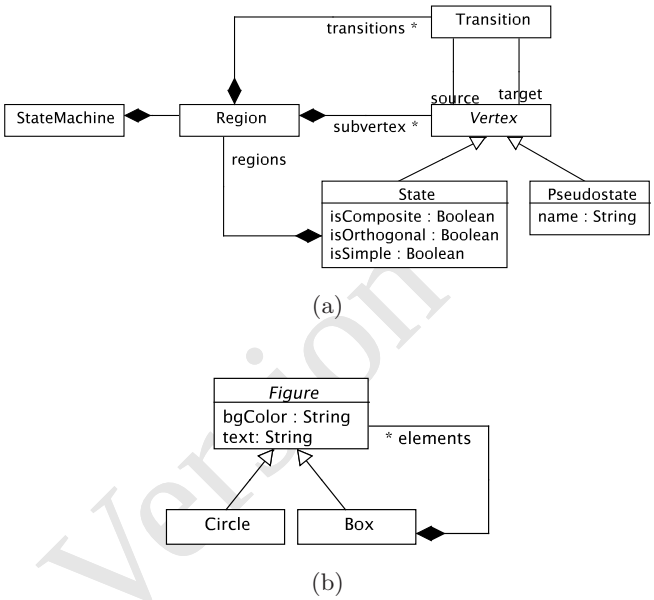

Figure 12 (a) Metamodel of a UML-like state machine. (b) Metamodel to represent graphical figures.

Figure 13 shows the result of a concrete transformation execution. The source model, in the left part, is a state machine composed of two orthogonal composite states, which has two possible configurations. The target model, in the right part, shows the generated graphics for each configuration.

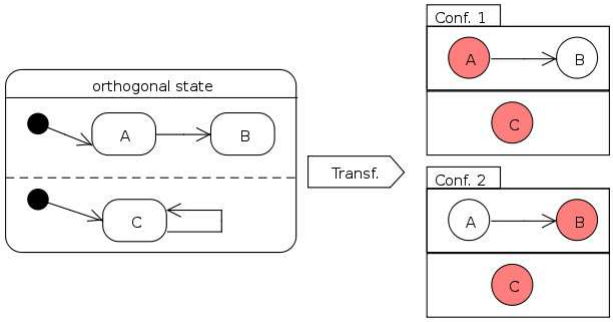

Figure 13 Result of the transformation from a state machine with two configurations. For each configuration a graphical representation of the active states is created.

In this transformation, the same state can be transformed several times, but depending on which configuration is being transformed (i.e. the current transformation context) the corresponding circle will be filled or not. For instance, state `A` is active in `conf1`, so it is filled, while in `conf2` it is not, so it is not filled. Since a state can belong to several configurations (for instance, state

C belongs to both conf1 and conf2) the rule transforming states to circles cannot know whether the circle must be filled or not: it needs to know the current configuration being transformed.

A possible solution is to implement a helper function that computes the configurations and creates annotated intermediate structures for each state of the configuration. For instance, a structure such as AnnotatedState { s : State, filled : Boolean } could be defined, and the rules would use AnnotatedState instead of State. The main problem of this approach is that the transformation logic is tangled within the helper function that computes the configurations, instead of being expressed in the form of rules. To avoid this tangling problem, we propose a phase-based solution.

The key point of our solution is the definition of a phase, named configurations, with a parameter which holds the current configuration being transformed. The execution DSL proposed in Section 3.2.2 is used to iterate over the collection of computed configurations, and to launch the configurations phase for each configuration, setting the value of parameter current_configuration to such a configuration. The following is the implementation of the transformation definition.

```
decorator UML::StateMachine
  def compute_configurations
    ...
  end
end

phase 'configurations' do
  parameter :filled_circle_color
  parameter :machine
  parameter :current_configuration

  rule 'statemachine2model' do
    from UML::StateMachine
    to   Figure::Model
    filter { |sm| sm == machine }
    mapping do |sm, figure_model|
      figure_model.figures = sm.regions
    end
  end

  rule 'region2box' do
    from UML::Region
    to   Figure::Box
    mapping do |region, box|
      box.elements = region.subvertex
    end
  end

  rule 'orthogonal2box' do
    from UML::State
    to   Figure::Box
    filter { |state| state.isOrthogonal }
    mapping do |state, box|
      box.elements = state.regions
    end
  end

  rule 'simple_state2circle' do
    from UML::State
    to   Figure::Circle
    filter { |state| state.isSimple }
    mapping do |state, circle|
      circle.text = state.name
      if current_configuration.include?(state)
        circle.bgColor = filled_circle_color
      end
    end
  end
end


scheduling do
  for machine in UML::StateMachine.all_objects
    for conf in machine.compute_configurations
      execute_phase 'configurations',
                    :filled_circle_color => 'red',
                    :machine => machine,
                    :current_configuration => conf

    end
  end
end
```

In this implementation, a decorator is used to add a method to compute the configurations of the state machine. The algorithm to compute the configurations is outside the scope of the paper, and it is only necessary to know that a configuration is encoded as a list of states (including composite states), so that the result of the compute_configuration is a list of lists.

The statemachine2model rule maps a state machine to a graphical model, that will contain the graphical representation of the configuration being transformed. Then, the region2box rule maps each region to a box which can contain other boxes (for orthogonal states) or circles (for simple states). It is important to notice that the simple_state2circle rule checks whether the state being transformed belongs to the current configuration (current_configuration.include?(state)), in order to set the circle color to value of the filled_circle_color parameter.

The scheduling code block traverses all state machines of the model (typically there will be only one), and computes the configurations of each. Then, for each configuration, the configurations phase is executed, passing as parameters the color for filled circles, the current state machine, and the current configuration (the notation => is used). The current configuration parame-

ter will be used as global information (context information) in the rules.

This example illustrates how the external information (i.e. current configuration) needed to transform a simple state can be collected without coupling the rule with a specific query logic. This approach has two advantages:

- The helper function (i.e. decorator) to compute the external information is more reusable since it does not need to annotate the result with information intended only for a specific transformation.
- The helper function is not invoked in the rule that uses it, but it is invoked externally to the rule (in the `execute` block). This makes the phase more reusable, since it is not coupled to a specific helper function. However, the mechanism would be more powerful if high order functions are supported in the underlying transformation language, so that the specific helper could be passed as a parameter, instead of just the result of the function call.

5.6 Internal transformation composition

As explained in Section 4.2 the phasing mechanism can be used as a foundation for internal transformation composition. The fact that a transformation definition is a kind of phase allows transformation definitions to be composed in the same way as phases.

Thus, it is possible to write a transformation definition as a reusable component, which can be adapted to different contexts by combining the transformation refinement mechanism and the importation of existing transformation definitions.

This example deals with the reuse of transformation definitions in the context of PIM-PSM [15] transformations. A UML class-model PIM will be transformed into a PSM describing a three-layer web architecture. In particular, the generation of bridges between layers will be addressed (usually bridges, when generated, are done at code level, but in this example they will be generated at model level). Generating bridges automatically alleviates the developer of the tedious task of writing the glue code between layers.

In the chosen platform, the view is implemented using Java Server Faces (JSF), the persistence layer uses the J2EE DAO pattern together with Hibernate, while the service layer is implemented using the Spring framework. The transformations between a UML class model representing the PIM and each layer (the PSM) are implemented in independent transformation definitions (i.e. files). Therefore, there are three existing transformation definitions we are interested in reusing:

- `web_jsf`. JSF Bean classes are generated for each UML class.
- `business_dao`. Business objects are implemented as POJO classes, which are made persistent by DAO classes. Each UML class has its corresponding DAO class.
- `spring_service`. A Spring service is generated for each UML class. This is a naive mapping, but in a more complex scenario, classes should be grouped into services. In any case, the approach to compose the transformations is the same.

Since these transformations have been developed to be reused, it is important to provide them with a specification of what they do. As we discussed, a transformation specification should be formed by an interface containing the trace relationships between source and target elements. For instance, the specification of the `web_jsf` transformation definition that a tool could extract from the actual transformation text is the following:

```
web-jsf:
  This transformation creates a set of managed
  beans from a definition of the UML classes.

 - trace relationships

    * UML::Model -> Web::JSF::Faces
      Each class model is mapped to a JSF
      configuration context

    * UML::Class -> Web::JSF::ManagedBean
      Each class is mapped to a managed bean.
      The scope of the managed bean is "request".

    * UML::Attribute -> Web::JSF::BeanProperty
      Each attribute will be a bean property, of
      the corresponding class.
```

From this specification, we know which target elements will be available when the transformation execution finishes. For instance, we can refine the managed beans to add new properties to them.

The following transformation definition imports and executes the three transformation definitions mentioned above. All these transformations work on the same source and target metamodels. In particular, the target metamodel is organized in three packages, namely, `JSF`, `Spring` and `Dao`. Then, the `integrate` phase is executed to merge the results of these transformations, in order to generate the bridges.

The `integrate` phase consists of two refinement rules, named `beans` and `dao_service`. The first one is intended to integrate Spring services with JSF, using a special kind of JSF bean. The `ServiceLocatorBean` is a bean that it is registered once in the application and will be used by the rest of beans to locate application services. In this way, the assignment `locator.services = Web::Spring ::Service.all_objects` gets all the Spring services and connects them to the locator object. The locator is also registered as a bean by the assignment

`faces.beans = locator`. Finally, a reference to the locator object is created for each existing JSF bean. To do this, the `trace_query` function is used to retrieve the proper JSF managed bean, a new property is added to the bean, and the locator is connected to such a property.

The second refinement rule, `dao_service`, connects the persistence implementation, in the form of DAO classes, to the Spring services.

```
phase 'integrate' do
  refinement_rule 'beans' do
    from UML::Model
    to   Web::JSF::Faces
    mapping do |model, faces|
      locator = Web::JSF::ServiceLocatorBean.new
      locator.scope = 'application'
      locator.name  = 'ServiceLocatorBean'
      locator.services = Web::Spring::Service.
                                  all_objects

      faces.beans = locator

      # Managed beans point to the locator
      model.classes.each do |klass|
        bean = trace_query(klass,
                        Web::JSF::ManagedBean)

        property = Web::JSF::ManagedProperty.new
        property.name     = 'serviceLocator'
        property.refValue = locator
        bean.properties   =  property
      end
    end
  end

  refinement_rule 'dao_service' do
    from ClassM::Class
    to   Web::Spring::Service,
         Web::Dao::DAOClass
    mapping do |klass, service, dao|
      property = Web::Spring::Property.new
      property.name     = dao.name
      property.refValue = dao
      service.properties = property
    end
  end
end

scheduling do
 import_transformation 'web_jsf.rb'
 import_transformation 'business_dao.rb'
 import_transformation 'spring_services.rb'
 execute_phase 'integrate'
end
```

The transformation definitions `web_jsf`, `business_dao`, and `spring_services` are reusable in other contexts, since they do not establish how they are related to other layers. The `integrate` phase is specifically written to connect the result of the transformations.

An issue that arises from this problem is when it is possible to compose the result of two transformation definitions. The result of a transformation execution is a target model conforming to a certain metamodel. Thus, if the composition of two transformations means merging or connecting their results, then there must be conformance at the target metamodel level. In the example, the three reused transformations use the same target metamodel, but in other scenarios different target metamodels can be considered. In this case, the metamodels can use the importation and inheritance facilities of the underlying meta-metamodel to achieve the required conformance.

6 Related work

6.1 Phasing in the literature

Phasing is mentioned in the feature model proposed in [4], where it 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 [3][24][10]. In OptimalJ [3] the transformation process is organized in a fixed number of phases, and the user has no control over the phasing mechanism. In [24], 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 [10], 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 only superficially described.

Regarding the ability of the phasing mechanism to deal with the complexity of model transformation, OptimalJ is an example of a tool which decomposes a transformation into phases to deal with complexity. However, the mechanism we propose is more general than the one used in OptimalJ where there are just three fixed phases, preventing the user from controlling the transformation process. Our proposal allows a flexible number of phases to be defined (in the sense explained in [24]), such that each transformation definition can define as many phases as needed. OptimalJ is a structured-oriented and target-driven approach [4][3], but our approach can be applied to source driven languages that rely on a more flexible rule organization.

6.2 Model transformation languages

The Tefkat model transformation language [12][13] provides a feature called *tracking classes* that allows named trace relationships between source and target elements to be explicitly set. These named relationships can be referenced from other rules, that can read the information placed in a tracking relationship, and even add more information. In our proposal, the trace relationships are always implicitly set by the transformation language. Also, when a trace relationship is referenced by a refinement rule, the source and target types are used, instead of naming the relationship. In any case, the intention of tracking classes and refinement rules is different. While tracking classes were designed to decouple rules and provide reuse at the rule level, phases and refinement rules have been designed to provide reuse at a coarser-grained level.

QVT operational mappings [1] allows chaining of transformation definitions written in any language, thus providing a means for external transformation composition. While it provides constructs to invoke transformations, it does not offer any composition operator apart from sequencing and parallelizing transformation executions. It also provides fine-grained mechanisms such as rule inheritance and mapping merge, which are useful to modularize large transformation definitions. Compared to our approach, the phasing mechanism not only allows us to chain transformation executions, but also to compose them, using refinement rules as a composition operator. Phase modularization is a coarse-grained mechanism, but in QVT modularization is done at the rule level.

With regard to graph-based approaches for model transformation, several scheduling mechanisms have been devised, such as abstract state machines [22], priorities [5] or layers [19]. Like phases, they are organizational mechanisms, which allow rule conflicts to be resolved. However, such mechanisms are not intended for modularity, and do not provide composition operators.

In [23] a DSL based approach intended to establish constraints in the combination of transformation definitions is presented. In our approach, it is up to the transformation developer to know which restrictions apply before composing transformation definitions. Therefore, this work is may complement our future research to find out a way to establish restrictions on the composition of transformation definitions.

6.3 Modularity and model transformation mechanics

The aspects in model transformation discussed in Sections 2 and 4 have been previously addressed in some works. Local-to-global and global-to-local transformations, in the context of translational semantics of DSLs, are discussed in [2], using ATL as the implementation language. Modularity in model transformation languages is addressed in [10][11]. In [10], modularity is studied in the context of adaptability of model transformations, while [11] studies whether current rule-based transformation languages provide proper modular constructs to write transformations with good properties, such as reusability and adaptability. In all these works, the main modular unit of the transformation languages is the rule, while we propose to use the phase as the main modular unit.

The phasing mechanism requires that dependencies between model elements in different phases were identified by the developer. Transformation languages such as ATL [8], relational QVT [16] or even RubyTL[18] provide declarative constructs to resolve dependencies automatically. However, as discussed in [2], these declarative approaches require that source elements producing target elements are found first. This may involve complex queries depending on the metamodel structure. Therefore, the dependencies are resolved by the developer at the source model level and the transformation engine translates them automatically to the target model. In our approach, dependencies can also be resolved at the target model level, relying on trace information by means of refinement rules, which make it easier to resolve the dependencies between model elements in some cases (as illustrated in Sections 5.4 and 5.5).

Finally, in [7] several aspects related to model transformation reuse are discussed, in particular seven ways of reuse are identified. In this context, our approach is intended for reuse in three ways: composition, specialization and parametrization.

7 Conclusions and future work

In this paper we have described a phasing mechanism which allows us to organize transformation definitions as an ordered set of phases. A phase is a modular construct that encapsulates a set of rules which perform a well-defined task.

We have analyzed the phasing mechanism from the perspective of several aspects related to the complexity of model transformations. First, we have shown how phases improve modularity, which promote reusability and adaptability of transformations. Also, we have described how phases help us to solve transformation problems involving global-to-local and local-to-global transformations.

Through several examples we have illustrated the usefulness of the phasing mechanism to tackle complexity by decomposition of a transformation in several subtasks, to read partially built target models in a consistent manner, to improve modularity of transformation definitions, avoiding tangling and scattering problems, to write transformation definitions involving local-to-global and global-to-local transformations, and to allow practical internal transformation composition to be achieved.

Another contribution of this work is that we provide an implementation of the mechanism in the RubyTL transformation language. Both the language, and the examples shown in this paper, can be downloaded from *http://gts.inf.um.es/age*.

As future work, it remains 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 whether there are other ways of composing phases in addition to using the internal trace of the transformation engine, as we have explained. In this way, a feature similar to the Tefkat tracking classes could be used as another way to compose phases.

With regard to improvements in the mechanism, the reuse of phases could be improved by providing some form of adaptors at the metamodel level, so that a phase can be adapted to use different metamodels from the ones it was designed for.

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. It would provide a mechanism to implement a single transformation with several languages whose transformation engines would rely on the same trace information to perform the transformation execution.

Acknowledgments

This work has been partially supported by Fundación Seneca (Murcia, Spain), grant 05645/PI/07. Thanks also to the reviewers for their insightful comments.

References

1. M. Belaunde. Transformation Composition in QVT. In *Proceedings of the First European Workshop on Composition of Model Transformations*, pages 45–52, July 2006.
2. T. Cleenewerck and I. Kurtev. Separation of concerns in translational semantics for dsls in model engineering. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 985–992, New York, NY, USA, 2007. ACM Press.
3. Compuware. OptimalJ, 2005. Available online: http://www.compuware.com.
4. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006.
5. J. de Lara and H. Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In *European Joint Conference on Theory And Practice of Software (FASE)*, pages 174–188, 2002.
6. R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, October 2004.
7. J. O. Goran K. Olsen, Jan Aagedal. Aspects of Reusable Model Transformations. In *Proceedings of the First European Workshop on Composition of Model Transformations*, pages 27–32, July 2006.
8. F. Jouault and I. Kurtev. Transforming models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.
9. A. Kleppe. MCC: A model transformation environment. In *2nd European Conference on Model Driven Architecture*, pages 173–187. Lecture Notes in Computer Science, 2006.
10. I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005. ISBN 90-365-2184-X.
11. 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.
12. M. Lawley, K. Duddy, A. Gerber, and K. Raymond. Language features for re-use and maintainability of MDA transformations. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, October 2004.
13. M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, 2005.
14. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997. MEY b 88:1 2.P-Ex.
15. Object Management Group. MDA Guide version 1.0.1. omg/2003-06-01, 2003. OMG document.
16. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005. www.omg.org/docs/ptc/05-11-01.pdf.
17. 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.
18. 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.
19. G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 446–453, 2003.
20. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
21. J. v. van Wijngaardeen and E. Visser. Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems. Technical report, Utrecht University, May 2003.

22. D. Varró and A. Balogh. The model transformation language of the viatra2 framework. *Sci. Comput. Program.*, 68(3):187–207, 2007.
23. D. Wageelar. Blackbox Composition of Model Transformations using Domain-Specic Modelling Languages. In *Proceedings of the First European Workshop on Composition of Model Transformations*, pages 21–26, July 2006.
24. 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.