Softw Syst Model (2010) 9:311–333
DOI 10.1007/s10270-009-0138-z

SPECIAL SECTION PAPER

Application of reflection in a model transformation language

Ivan Kurtev

Received: 10 November 2008 / Revised: 2 September 2009 / Accepted: 24 September 2009 / Published online: 10 November 2009

Abstract Computational reflection is a well-known technique applied in many existing programming languages ranging from functional to object-oriented languages. In this paper we study the possibilities and benefits of introducing and using reflection in a rule-based model transformation language. The paper identifies some language abstractions to achieve structural and behavioral reflection. Reflective features are motivated by examples of problems derived from the experience with currently used transformation languages. Example solutions are given by using an experimental language with reflective capabilities. The paper also outlines possible implementation strategies for adding reflection to a language and discusses their advantages and disadvantages.

Keywords Reflection · Model transformation languages · MDE · MISTRAL

1 Introduction

Computational reflection is a technique applied in many programming languages to solve non-trivial problems. Usually, adding reflective capabilities to a language is motivated by the need to improve certain quality attributes in the programs such as runtime adaptability, long-term maintainability, modularity and composability.

Communicated by Jeff Gray.

I. Kurtev (✉)
Software Engineering Group, University of Twente,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
e-mail: kurtev@ewi.utwente.nl; ivan.kurtev@gmail.com

Reflection may be applied to languages based on different paradigms. It was first proposed by Brian Smith in the context of Lisp [27] and later was successfully introduced in object-oriented languages [21] such as Smalltalk, CLOS, and Java [4].

The problems that may be tackled by using reflection span a wide spectrum. Here we briefly mention some literature sources and the issues they address. It was shown that *debugging* and *tracing* can be based on reflective techniques [20]. The Composition Filters approach to *object composition* [2] is based on a restricted reflection on message passing between objects. The application of reflection in *aspect-oriented programming* is studied in a number of works [26,28]. A limited form of reflection (*introspection*) was introduced in the Java language. The reflective API available in ECore [3] allows building *generic model editors*. Kiczales et al. [13] use reflection to adapt an existing language (CLOS) to user-specific needs, thus allowing a reflective language to be treated as an open-end, *customizable language*, i.e., as a family of related languages.

The ability of reflection to be applied across diverse types of languages to solve a significant number of problems is appealing. The quality characteristics that a reflective program may possess (e.g. *adaptability* and *reusability*) are often required in model transformation specifications. Therefore, it is worth studying how reflection may be applied in current model transformation languages in the context of Model-Driven Engineering (MDE). Our study is also motivated by problems that are difficult to solve with the available techniques in the majority of these languages.

Transformation languages already provide a solution for adapting transformation specifications to certain needs when the language constructs are not expressive enough. This solution is based on *higher-order transformations* (HOT) [30]. In several MDE approaches, transformation languages

are defined by a metamodel and therefore transformation specifications are models. Transforming these specifications is not different from any other transformation executed on models. A higher-order transformation transforms transformations to other transformations. However, this solution is static. It accesses only one type of computational objects: the transformation programs, and allows changes only before their execution. A fully-fledged reflection mechanism goes beyond this. It allows accessing and altering runtime computational objects and changing programs at runtime.

In this paper we study the possibilities for adding reflection to an experimental model transformation language called MISTRAL [16]. The main challenge is to balance between the expressiveness and the safety of the reflective features. We are driven by a set of problems encountered in our experience with current transformation languages. The introduced reflective features aim at solving these problems in a concise manner that improves the quality of the transformation specifications. Furthermore, the presence of reflection allows adaptation of the language without extending its concrete syntax and re-implementing its engine.

The most important problems that motivate this work are: achieving flexible trace generation in model transformations, fine-grained control over the rule execution order, improving change impact analysis and change propagation when source or target models change, and achieving better composability and adaptability of existing transformation specifications. We also aim at a conceptual solution that is applicable to both imperative and declarative transformation languages. We present a set of computational objects and events that are exposed to metaprograms. Examples are presented to illustrate the motivation and the applicability of the proposed reflective features.

The paper is organized as follows. Section 2 gives background knowledge about reflection and a general view how reflection can be applied in model transformation languages. Section 3 shows several motivating examples that benefit from the availability of reflection. An experimental transformation language named MISTRAL is presented in Sect. 4. Section 5 systematically identifies reflective features for MISTRAL based on its execution semantics. Section 6 gives solutions to the motivating examples. Section 7 discusses the experienced difficulties, the implementation options, and the limitations of our work. Section 8 presents related work. Section 9 concludes the paper and outlines future work.

2 Computational reflection

We present a brief theoretical overview on reflection and the relevant concepts in Sect. 2.1. The concept of reflection is applied to the domain of model transformation languages. We make assumptions about the structures and operations available in a model transformation execution environment that make our approach general enough and potentially applicable to other languages (Sect. 2.2). Section 2.3 introduces a two-dimensional space that serves as a guiding framework for designing reflective infrastructures.

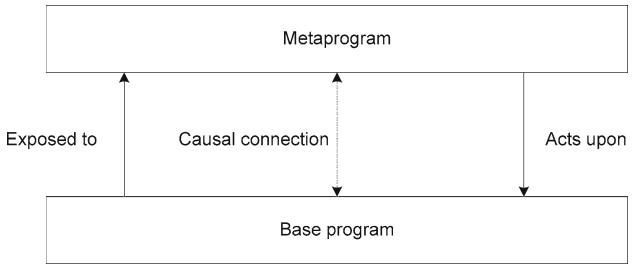

Fig. 1 Relations between the base program and metaprogram

2.1 Background

Reflection is a capability to perform computation about another computation. A computation is an execution of a program written in a programming language. A running program may expose some elements of its execution environment via some interface. It is said that the running program resides at the *base level* of computation and is called the *base program*. Another program may be written that accesses and eventually changes the exposed elements of the execution environment for the base program. Such a program is called a *metaprogram* and its computational environment is called the *metalevel*[1]. The relation between the base level and the metalevel is causal [21]. This means that changes in the base level are visible in the metalevel and changes made in the metalevel affect the base level. Figure 1 illustrates the relations between the base program and the metaprogram.

The concept of reflection was proposed by Brian Smith [27] and later elaborated for a number of languages. Reflection was intensively studied in the context of object-oriented languages. In this paper we use terminology from this field. A presentation of the original work of Smith is given in [8].

In his PhD thesis, Tanter [29] presents an overview of existing reflective systems and proposes a set of definitions for reflective mechanisms. He identifies two orthogonal classification schemes for reflection. They are based on the ability of the metaprogram to access various elements of the base level and the ability to alter the base level. The classifications are based on the concepts of *structural*

[1] The term *metalevel* used in this paper is different from the same term used to denote a layer in a metamodeling architecture.

reflection, *behavioral reflection*, *introspection*, and *intercession*. We repeat the definitions from [29] in order to establish a set of concepts for this paper.

Definition 1 (*Structural Reflection*) The ability of a program to access a representation of its structure, as it is defined in the programming language.

Definition 2 (*Behavioral Reflection*) The ability of a program to access a dynamic representation of itself, that is to say, of the operational execution of the program as it is defined by the programming language implementation (processor).

Definition 3 (*Introspection*) The ability of a program to reason about reifications of otherwise implicit aspects of itself or of the programming language implementation (processor).

Definition 4 (*Intercession*) The ability of a program to act upon reifications of otherwise implicit aspects of itself or of the programming language implementation (processor).

It should be noted that introspection allows only reasoning on reifications, that is, it is *read-only* form of reflection, whereas intercession allows acting upon reifications including possible changes.

Definitions 3 and 4 use the important concept of *reification*. Reification is defined as follows.

Definition 5 (*Reification*) The process by which the state of the running program is passed to the program itself, suitably packaged (reified) so that the program can manipulate it.

An example of reification is the process of explicit representation of a message sent from one object to another object in an object-oriented system. The reified message is a structure that contains the sender and the receiver objects, the name of the method responsible for handling the message and the message parameters (if any). Such a reified message may be a subject of introspection and intercession. Introspection and intercession may be applied in both structural and behavioral reflection. For example, intercession may affect both the structural part and the behavioral part of the base level. Concerning the structural part, the metaprogram may change the structure of the running program. Concerning the behavioral part, the metaprogram may change the runtime structures of the running program and change its behavior at runtime. A reified message, for example, may be manipulated by changing the name of the method and then passed to the receiver. In this way, the intercession allows the altering of the default message passing behavior.

We consider all types of reflection in this paper: structural, behavioral, introspection, and intercession.

2.2 Towards a common model of execution of model transformation programs

The success of applying reflection in object-oriented (OO) languages is due to the possibility of finding a common computational model valid for a large set of OO languages. A minimal computation model consists of objects that exchange messages. The typical representation of an object-based execution environment includes interfaces to objects, reification of messages and other computational events.

The following assumptions lead to a concise yet unified computational model for execution of model transformations:

- The transformation language operates on models that conform to metamodels. The transformation language by itself is defined by a metamodel;
- The language is rule-based. Transformation specification consists of transformation rules executed on tuples of source elements. Every rule may create new elements, update existing elements, and delete elements. Creation and update involve setting property values.

The majority of transformation languages used in MDE satisfy the assumptions. Many transformation languages operate in an ECore or MOF based metamodeling environment. The abstract syntax of the language may be expressed in a metamodel. We should note that there are transformation languages that are not based on MOF/ECore. For example, many graph transformation languages do not rely on the OMG standards (including MOF).

Furthermore, a transformation rule is the most commonly found modular unit that operates on the model elements. It should be noted that it is possible to have transformation approaches not organized around the rule concept. Such approaches are usually built on a general-purpose language and a library for model manipulation. For example, the language SiTra follows this approach [1].

A transformation may be considered as a sequence of events. Execution of a rule is such an event. Execution of a rule involves other events: identifying matches of the rule source and executing the effects of the rule per match. Effects are creation, deletion, and update of model elements. These events may occur in various orders. For example, in ATL [10], a declarative transformation is executed in the following order: matching of all the rules, for every match the creation of new elements is executed, and property values are assigned. In the case of an imperative language such as QVT Operational Mappings [24] the order of matching, creation, and property value assignment may differ. Regardless of the order of events, however, the set of event types is stable across the languages.

Table 1 Example operations on transformation specification and transformation rule

	Structural reflection	Behavioral reflection
Introspection	*Transformation specification*	*Transformation specification*
	Read the transformation specification	Check the status of the transformation (started, in execution, executed)
		Obtain the currently executed rule
	Transformation rule	*Transformation rule*
	Read the rule structure	Check the rule status
	Read data related to a rule match and execution (source element, created target elements)	Check the current match being executed
Intercession	*Transformation specification*	*Transformation specification*
	Change the transformation specification at runtime	Start/stop transformation execution
	Transformation rule	*Transformation rule*
	Change the rule structure at runtime	Execute rule Mark rule as executed

This is our starting point for identifying the elements that should be exposed during a model transformation execution.

2.3 Reflection in the common execution model

McAffer [22] proposes two approaches for identifying the elements that should/could be available at the metalevel. The first approach considers the language constructs that should be exposed. This corresponds to identifying the elements accessible in structural reflection. The second approach considers the events observed during the program execution. This corresponds to identifying the elements accessed in behavioral reflection.

Clearly, both approaches may be combined. They also span the dimension of introspection/intercession. The two dimensions give a large solution space for designing a reflective infrastructure ranging from a limited *structural introspection* to a difficult to design and control *behavioral intercession*.

The paper gives an example of possible structures and operations at the metalevel by considering two constructs found in transformation languages and the relevant events: *transformation specification* and *transformation rule*. We place them in a two-dimensional space where the first dimension denotes the structural/behavioral dichotomy and the second dimension denotes the introspection/intercession dichotomy. The space consists of four points shown as cells in Table 1. Each cell provides transformation and rule constructs. Constructs are shown in italics. A bulleted list indicates possible operations on them.

Table 1 shows that even for two constructs with only part of the possible operations we have 12 operations (see the bulleted elements).

The infrastructure for a metalevel usually allows intercepting the relevant events in the execution (behavioral introspection) and reification of the relevant data as *metaobjects*. The term *metaobject* should be interpreted in the broad sense as a data structure describing another data structure (not necessarily implemented in an OO language). Metaobjects may be queried by the metaprogram, may be changed (example of structural intercession), and operations may be applied on them that eventually affect the program behavior (behavioral intercession).

In the remaining part of the paper we will identify the elements of a metalevel (similarly to Table 1) for our experimental language. The identification is guided by a set of motivating scenarios and the analysis of the execution semantics of the language. We present the motivating scenarios in Sect. 3. Section 4 presents MISTRAL, which is a transformation language that has reflective capabilities.

3 Motivating scenarios

Our motivating scenarios serve as use cases for the initial identification of reflective features. The first use case addresses a common problem in model transformations: generation of custom trace links. The second use case is derived from our experience in implementing aspect weavers as model transformations.

3.1 Example: generation of custom trace links

Some transformation languages and their engines create trace links between source and target elements. These links allow identification of the target elements obtained from a given source element by applying a given rule. Often, the links are maintained in the proprietary internal structures of the transformation engine and are not available after the execution of the transformation. However, there may be cases in which the transformation developer prefers to create their own trace structures. This problem is already analyzed in [11,17]. It is desirable to create a generic trace generation functionality that is independent from the concrete transformation specification and may be reused in multiple transformation executions.

Consider a transformation that transforms UML class models to Java programs. We require that every UML attribute is transformed to a private field and two methods for getting and setting the value of the field. For brevity we do not give the metamodels of UML and Java. The following pseudo code shows a part of the transformation specification. The code specifies rules with source patterns and target actions. Actions in this case are instantiations of target elements that assign values to their properties. Source and target elements are assigned to variables (e.g. "sp", "tp", "f", etc.). These variables are used to refer to the elements in expressions, for example, in line 9. No execution order among the rules is assumed. OCL is used to query the source models.

```
1.  Rule copyPackage {
2.    source [sp : UML!Package]
3.    target [tp : Java!Package {name = sp.name}]
4.  }
5.
6.  Rule copyClass {
7.   source [sc : UML!Class]
8.   target [tc : Java!Class {name = sc.name,
9.               fields = sc.attribute->collect(a | transformation.trace(a, 'f')),
10.              methods = sc.attribute->collect(a |
11.                   Sequence{transformation.trace(a, 'getter'),
12.                            transformation.trace(a, 'setter')})->flatten()}]
13. }
14.
15. Rule transformAttribute {
16.   source [sa : UML!Attribute]
17.   target [f : Java!Field{name  = sa.name, isPublic = false,
18.                          type = transformation.trace(sa.type, 'tc')},
19.           getter : Java!Method{name  = 'get_'+sa.name, isPublic = true},
20.          setter : Java!Method{name  = 'set_'+sa.name, isPublic = true}]
21. }
```

We aim at producing an output model that contains trace links between source and target elements. The metamodel for traces expressed in KM3 [12] is given below. KM3 is a concrete textual syntax for expressing ECore metamodels. The syntax strongly resembles the Java syntax and the syntactical elements corresponds almost directly to the metaclasses in ECore.

```
1.  package Traces{
2.
3.   class Trace {
4.    attribute ruleName : String;
5.    attribute sourceName : String;
6.    attribute targetName [1-*]: String;
7.   }
8.  }
9.
10. package PrimitiveTypes {
11.  datatype String;
12. }
```

The idea is that for every execution of a rule over a source node we create an instance of class *Trace* that contains the name of the rule, the name of the source element, and a list of the names of the target elements. This may be done by introducing a new instantiation in every rule that uses the identifiers of the source and target elements per rule. The following code shows the first two rules extended with trace generation functionality (lines 4–6, 14–16).

```
1.  Rule copyPackage {
2.    source [sp : UML!Package]
3.    target [tp : Java!Package {name = sp.name},
4.            trace : Traces!Trace {ruleName = 'copyPackage',
5.                                  sourceName = sp.name,
6.                                  targetName = sp.name)}]
7.  }
8.
9.  Rule copyClass {
10.   source [sc : UML!Class]
11.   target [tc : Java!Class {name = sc.name,
12.               fields =sc.attribute->collect(a | transformation.trace(a, 'f')) ,
13.               ..........
14.           trace : Traces!Trace {ruleName = 'copyClass',
15.                                 sourceName = sc.name,
16.                                 targetName = sc.name)}]
17. }
```

This solution, however, has two problems that reduce the *reusability* and *adaptability* of the transformation specification. First, the trace generation functionality depends on the context of a particular rule and is repeated in every rule. This hinders the reusability of this functionality since it is not specified in a single language module. Second, the adaptability of the transformation specification is deteriorated since the trace generation cannot be added and removed without manually changing the transformation specification.

To solve the first problem we need a mechanism for abstracting the trace generation functionality from the details of a particular rule. For this purpose we need to obtain the name of the context rule and to obtain details about the current match of the rule and the currently created target elements. *Structural introspection* may help in getting the name of the rule. *Behavioral introspection* may help in obtaining the information about the context of the rule at runtime (the source element being matched and the current target elements).

To solve the second problem we need to modify the rules before their execution by adding a new action: creation of trace records. This may be done by extending the language with a dedicated composition construct, e.g. rule inheritance. Another option is to use *structural intercession* that allows changing the transformation specification at runtime. For example, a transformation may be changed before its execution. The change may insert the trace instantiations to a set of selected rules (or all rules). An example of this is given in Sect. 6.1.

3.2 Example: controlling aspect weaving at shared join points

This scenario is inspired by a problem in Aspect-Oriented Programming (AOP) [14]. Aspects are constructs that modularize code scattered across multiple locations in a program. They are inserted in designated points (called *join points*) by a process called *aspect weaving*. In general, aspects do not know about each other and more than one aspect may be woven in a single join point. Possibly, aspect interference may occur. This problem is known as *weaving of aspects at shared join points* [7]. Usually, a solution requires some policy of ordering of the aspect weaving.

In this scenario we perform aspect weaving by executing model transformation rules on a source Java program. Consider the following two rules that implement a logging aspect and a synchronization aspect.

```
1.  Rule addLogAspect {
2.    source[sourceMethod : Java!Method condition{(sourceMethod.name = 'methodA')
3.                                          or (sourceMethod.name = 'methodC')}]
4.    target[logInvocation : Java!MethodCall{variableName='Log',
5.                                          methodName = 'log'},
6.           update sourceMethod{statements =
7.                Sequence{logInvocation}->union(sourceMethod.statements)}
8.          ]
9.  }
10.
11. Rule addSynchAspect {
12.   source[sourceMethod : Java!Method]
13.   target[obtainLockInvocation : Java!MethodCall{variableName = 'Lock',
14.                                                 methodName = 'getLock'},
15.          releaseLockInvocation : Java!MethodCall{variableName = 'Lock',
16.                                                  methodName = 'releaseLock'},
17.          update sourceMethod{statements =
18.         Sequence{obtainLockInvocation}->union(sourceMethod.statements)->union(
19.                                          Sequence{releaseLockInvocation})}
20.         ]
21. }
```

Rule *addLogAspect* creates a call to a method that implements the logging functionality (lines 4–5). Then the call is inserted before the statements of the selected Java methods. The rule weaves the aspect in methods in the source program named *methodA* or *methodC* (see the rule condition in lines 2–3). The weaving is done by using an *update* action that modifies the statements of the source element (line 6).

Rule *addSynchAspect* inserts two method calls in an existing method. The first one is inserted in the beginning of the method body and is responsible for obtaining a synchronization lock (line 13). The second one is inserted at the end of the method body and releases the lock (line 15). This aspect is woven in all methods of the source program (see the source of the rule on line 12).

The two aspects have shared join points. The question is in which order the aspects are applied at these shared join points. This problem is studied in [15] and several solutions are proposed. Assume that for some Java methods we want to apply first the logging aspect and then the synchronization aspect while for other methods we reverse the order. We require that the aspects are reusable and remain unaware about each other. The order of weaving should be specified separately. In most declarative transformation languages, however, the order of rule execution cannot be controlled. Even with the possibility to order the rule execution, the problem cannot be completely solved. Rule ordering ensures that a rule is executed before another rule for *all* matched source elements,whereas we require finer control at the level of a *single* source node. If the language is imperative, controlling the lead rule order may to reimplementation of the code responsible for the execution order.

The problem may be solved if fine grained control over rule execution is available. The programmer should be able to inspect the rules that match a single source element and impose partial order over rule execution. These features are examples of *behavioral reflection*.

4 MISTRAL: an experimental language with reflective features

We present an experimental transformation language with reflective features. The language syntax is presented informally by giving an example in Sect. 4.1. Section 4.2 gives the pseudo code of the execution algorithm for MISTRAL transformations. The algorithm is a starting point for designing the reflective features of the language.

4.1 Example MISTRAL program

MISTRAL was initially described in [16] without having a complete execution engine. Recently we implemented an experimental interpreter for the language. This section briefly describes the language constructs and their meaning.

Consider the following code fragment that is used in a larger transformation for flattening class hierarchies in UML class models that use single inheritance. Only the identification of attributes of a class (including the inherited attributes)

is shown. Creation of the target classes is not shown. The full version of the transformation is available from [23].

```
1.  transformation Flattening
2.  input s : UML
3.  output t : UML
4.
5.  allAttributesOfRoot ModelElementRule {
6.    source [class : UML!Class condition{class.extends->isEmpty()}]
7.    target [attr : Sequence(UML!Attribute) = class.attribute]
8.  }
9.
10. allAttributesOfNonRoot ModelElementRule {
11.   source [class : UML!Class condition{class.extends->size() > 0}]
12.   target [attr : Sequence(UML!Attribute) = class.attribute->union(
13.                     transformation.trace(class.extends->first(), 'attr'))
14.         ]
15. }
```

A transformation declares a number of input and output models that are assigned to variables typed by the corresponding metamodel (lines 1–3). Output models do not exist in advance and are created during the execution of the transformation. The input models may be changed at runtime.

Every transformation contains named model element rules (*allAttributesOfRoot*, *allAttributesOfNonRoot*). Model element rules have a source that is identified by a single variable of a given type and an optional condition over the values of the variable. The purpose of a model element rule is to execute actions enumerated in its target. Actions are executed for every match of the rule source. Two types of actions are supported: *instantiation* and *update* (update is not shown in this example). An instantiation action causes creation of new elements. The types of the elements are types from a metamodel and the built-in OCL types (for example, *Integer*, *Boolean*, *Sequence*, *Set*).

The example code determines the total set of attributes per class including the locally defined and the inherited attributes. Rule *allAttributesOfRoot* determines the attributes of root classes. The total set of attributes is represented in the target of the rule as variable *attr* of type *Sequence*. It has an initialization expression. Rule *allAttributesOfNonRoot* determines the attributes of classes that extend other classes. The set of the attributes is the union of the locally defined and the inherited attributes. The inherited attributes are obtained by navigating to the parent class and invoking the trace resolution function that returns target elements for a given source. The resolution function is called 'trace'. It accepts two arguments: the source element and the identifier of the target element. An example invocation is given in line 13.

The current implementation of the language supports rules that match only to a single source element. A source element may be matched by multiple rules. Target model elements are navigable during the transformation execution. Properties of model elements are called *slots*. If a slot value of a target element is requested but not yet assigned, the value will be calculated on demand and assigned.

MISTRAL is a DSL (Domain-Specific Language) aimed at specifying unidirectional model transformations. It encourages a declarative specification style where relations between source and target elements are specified. The developer does not encode the rule execution order. The rule execution order and the action execution order within a rule are decided by the interpreter at runtime. As explained in [16], if a transformation does not change the models by using update and delete actions the transformation execution is deterministic. This means that different execution orders produce the same output.

The declarative nature of the language limits its expressive power. There are no constructs for iteration and condition that can be used to specify sophisticated rule ordering. Declarativeness and domain-specificity are major design decisions for MISTRAL. Section 5 shows how these decisions negatively affect the design of the reflective architecture.

4.2 Execution semantics

We give the execution semantics of MISTRAL in a pseudo code that presents the execution algorithm for a

transformation. The starting point is the procedure *Execute-Transformation* that takes a transformation *t* and a map of *models*.

```
1.   ExecuteTransformation(Transformation t,
2.                         Map models) {
3.
4.      Traces traces = new Traces();
5.
6.      ForEach rule in t.rules {
7.         MatchRule(rule, models, traces)
8.      }
9.
10.     ForEach rule in t.rules {
11.        ExecuteRule(rule, models, traces)
12.     }
13.  }
```

We first create the traces (line 4) as an internal structure that keeps track of the correspondences between source and target elements and the rule that generates the target elements from a given source. The first step in the execution is to match the source patterns of all the rules over the source models (lines 6–8). The matching initializes the traces. The second step is to execute all the rules by using the models and traces (lines 10–12).

The procedure *MatchRule* finds all the matches for the source pattern of a given rule.

The matching is done over a single source model identified by the *extentName* property of *RuleSource* (line 18). All the model elements of a given type are obtained and then filtered by checking the rule source condition (lines 20–25). Traces are updated accordingly (line 23).

The procedure *ExecuteRule* iterates over all the matches for a given rule. Since the transformation of a given source element may have been already performed on demand by another rule, the algorithm transforms only the source elements not transformed yet (line 34).

```
14.  MatchRule(Rule rule,
15.            Map models,
16.            Traces traces) {
17.
18.     Model sourceModel = models.get(rule.source.extentName)
19.
20.     List elements = sourceModel.getAllInstances(rule.source.type)
21.     ForEach el in elements {
22.        If el satisfies rule.source.condition {
23.           traces.addMatch(el, rule)
24.        }
25.     }
26.  }
```

```
27.  ExecuteRule(Rule rule,
28.               Map models,
29.               Traces traces){
30.
31.     List matches = traces.getMatchesForRule(rule)
32.
33.     ForEach match in matches {
34.        If match not transformed {
35.           ExecuteRuleOnMatch(rule, match, models, traces)
36.           Mark match as transformed
37.        }
38.     }
39.  }
```

The procedure *ExecuteRuleOnMatch* generates the target elements for a single source element and a given rule that matches this element. The procedure iterates over the actions specified in the rule (lines 45–56). In order to simplify the explanation we consider only instantiation actions. At the moment of writing this paper, actions that update and delete existing elements are implemented partially and are in an immature form. We did not consider them as a part of the reflective framework.

If a given action is not performed yet (this is checked by inspecting the traces, line 46) an element is created and added to the target model (lines 47–49). Traces are updated accordingly (line 50). After creating the target element, its slots are assigned with values (lines 52–54).

The procedure *AssignSlotValue* calculates the values of slots and takes care of possible circularity dependencies among slot values. Assume that the value of a slot A requires the value of a slot B, which in turn requires the value of A. In this case there is a mutual dependency between the values and both cannot be calculated. In this respect the execution of a MISTRAL transformation is similar to the calculation of an attribute grammar [25]. The cyclic attribute grammars cannot be executed. MISTRAL transformations with such cycles cannot be executed likewise. Cycles are detected at runtime.

During the transformation execution, every slot of a given model element may be in three possible states: *not processed*, *processing*, and *processed*. The procedure first checks if the

```
40.  ExecuteRuleOnMatch(Rule rule,
41.                     Element match,
42.                     Map models,
43.                     Traces traces){
44.
45.     ForEach action in rule.target {
46.       If not traces.containsTarget(match, rule, action.id) {
47.          Model targetModel = models.get(action.extentName)
48.          Element result = create target element of type action.type
49.          targetModel.add(result)
50.          traces.update(match, rule, action.id, result)
51.
52.          ForEach slot in action.slotAssignments {
53.           AssignSlotValue(rule, match, result, slot, models, traces)
54.         }
55.       }
56.     }
57.  }
```

slot is in the processing state (line 65). If so, this means that the process of value calculation has started but is not finished yet. The calculation is trying to obtain other slot values (recall that in MISTRAL the target models are navigable) and the value of the current slot is requested again before the calculation is complete. In this case a cycle is detected and an error is thrown (line 66).

If no cycle is detected yet, the initialization expression in OCL is evaluated (line 71). The expression may request values of other slots or may try to obtain target elements from their corresponding source (see the further explanation of the procedure *Resolve*). The method *evaluateExpression* invokes the OCL interpreter of MISTRAL.

In the context of the MISTRAL language, OCL is extended with an operation that resolves traces from source to target elements (see line 13 in the example). In order to evaluate the operation call, the OCL interpreter invokes the procedure *Resolve*. This invocation is not shown since we abstract from the details of the interpretation of OCL. The procedure *Resolve* returns the target model elements obtained from a given source by applying some transformation rule. Target elements are assigned with an identifier. If the identifier is not known an error is generated (line 84). The target element may be already created. In this case it is obtained from the traces and returned (line 91). If it is not created then the required transformation rule is executed on the given source on demand (line 88).

```
58.  AssignSlotValue(Rule rule,
59.                  Element match,
60.                  Element target,
61.                  SlotAssignment slot,
62.                  Map models,
63.                  Traces traces){
64.
65.    If slot is in processing state {
66.       Throw Error("Circularity among slot values")
67.    }
68.
69.    If slot is not processed {
70.       Mark slot as in processing state
71.       List result = slot.expression.evaluateExpression(match,
72.                                                        target,
73.                                                        models,
74.                                                        traces)
75.       target.set(slot.name, result)
76.       Mark slot as processed
77.    }
78.  }
```

```
79.  Resolve(Element source,
80.           String identifier,
81.           Traces traces, Map models){
82.
83.      If not traces.existsIdForSource(source, identifier) {
84.         Throw Error("There is no element identified by identifier")
85.      }
86.
87.      If traces.getTargetForSource(source, identifier) is null {
88.         ExecuteRuleOnMatch(traces.getRule(source, identifier),
89.                                              source, models, traces)
90.      }
91.
92.      Return traces.getTargetForSource(source, identifier)
93.
94.  }
```

5 Adding reflection to MISTRAL

We illustrate how the execution algorithm and the motivating scenarios help in identifying the structures available at the metalevel (Sect. 5.1). Section 5.2 presents the extension of MISTRAL with reflection.

5.1 Identification of relevant computational objects and events

Deciding on reflection features of a language is not an easy task. The challenge is to identify the computational objects and events that should be exposed to the metaprograms. One approach is to provide a rich set of features covering most of the language constructs and execution structures. Another approach is to select a set of concrete problems and to choose only the necessary features to solve them. We chose a combination of both approaches by analyzing the potential reflective features and selecting mainly those for which we can identify and envisage a motivating scenario. Following McAffer [22], we examine thoroughly the already presented execution semantics in order to identify possible computational objects and events. Some of them are included in the MISTRAL reflective framework.

Table 2 gives a summary of our analysis of potential features and what has been selected.

5.1.1 ExecuteTransformation

A major decision is not to include the trace object in the reflective framework. It is possible to fix an abstract interface to such an object and to let the developer provide their custom implementation. This can solve the problem in the first scenario (generating custom traces). We decided to solve this problem by other means illustrated in Sect. 6. Furthermore, the decision to match all the rules before their execution is important for the execution algorithm and has a positive impact on the performance during transformation execution. We decided not to provide a reflective access to the rule matching event.

Transformation specification will be exposed to metaprograms both for reading and writing. Changing a transformation at runtime allows adding new rules and making adaptations of the rules.

The default execution algorithm does not assume rule execution order. The first scenario shows a need for explicit execution order. We provide a possibility to customize the rule execution order by handling the event of starting the rule execution.

5.1.2 MatchRule

As explained above the matching process is not accessible from metaprograms. Therefore, this procedure does not provide reflective features.

5.1.3 ExecuteRule

The main responsibility of this procedure is to iterate over the matches for a single rule and to invoke the procedure *ExecuteRuleOnMatch*. A possible alteration of the default behavior

Table 2 Reflective features in MISTRAL

Procedure	Possible computational objects and events	Included computational objects and events
ExecuteTransformation	Objects:	Objects:
	Transformation specification	Transformation specification
	Traces	
	Events:	Events:
	Transformation execution	Transformation execution
	Trace creation	Rule execution
	Rule matching	
	Rule execution	
MatchRule	Objects:	None
	Rule	
	Events:	
	Match of a rule	
ExecuteRule	Objects:	None
	Rule	
	Current status of rule	
	Events:	
	Execution of matches	
ExecuteRuleOnMatch	Objects:	Objects:
	Rule	Rule
	Current match	Current match
	Rules per match	Rules per match
	Events:	
	Execution of target actions	
AssignSlotValue	Objects:	Objects:
	Rule	Rule
	Current match	Current match
	Current slot	Current slot
Resolve	Objects:	Objects:
	Source element	Source element
	Identifier	Identifier

is to provide an order for handling the matches. Currently, we do not see a convincing usage for this that goes beyond the slogan "nice to have". This procedure does not contribute to the reflective framework.

5.1.4 ExecuteRuleOnMatch

This procedure performs important tasks during the transformation execution. It provides information about the current execution context of a rule. The possibility to react to the event of execution and to read/change the execution context is included in the reflective framework. For example, a metaprogram may perform recording of the execution order, or generate custom traces by handling the event when this procedure is executed. Furthermore, a rule may be changed before its execution. This feature will be used in Sect. 6.1 to solve the trace generation problem.

The execution of target actions is the main event in this procedure. Currently only the instantiation and update actions are supported. The execution order among them is automatically determined.

5.1.5 AssignSlotValue

This procedure is called in the execution of a rule target action. The metaprograms are notified before the execution of the procedure and get an access to the execution context: the enclosing rule, the current match of this rule and the information about the slot. The reflective mechanism is able to change the contextual information before resuming the default execution semantics for assigning slot values.

5.1.6 Resolve

This procedure implements the algorithm for mapping source to target elements based on the internal trace structure. When invoked, the metaprogram may handle the invocation event and to access its context: the source element and the identifier of the target element. The context may be changed, thus allowing the metaprograms to alter the default resolution algorithm. It should be noted that the metaprograms are not allowed to change the internal trace structure but may read it. The capability to reflect upon the resolution algorithm improves the composability of independently developed transformation specifications.

5.2 Expressing metaprograms

The base program and the metaprogram can be possibly expressed in different languages. However, metaprograms are often written in the same language used at the base level. This is the case for the metaprograms for Java, Smalltalk and CLOS reflective frameworks, for example. An attempt to apply this approach to MISTRAL faces a major obstacle. MISTRAL is a DSL aimed at specifying model transformations. However, some computations at the metalevel require more expressive power that only a general-purpose language may provide.

In the general case, we cannot expect that all metaprograms acting upon a program expressed in a DSL can be expressed in this DSL. Consider the example when the rules that match a single source element need to be executed in a particular order. We assume that the metaprogram is provided with a reified structure that contains a list of matching rules for a given element. The metaprogram has to reorder part of the rules before handing in the execution to the base level. Doing this in a rule-based language not armed with constructs for iteration and condition is difficult and may be even impossible.

This obstacle prevents the usage of the MISTRAL language in its current form to express some metaprograms. Therefore, the language has to be extended with new features that allow expressing the required metaprograms. Another option is to express the metaprograms in a general-purpose language.

In this paper we opt for the first alternative. We extend the language with new types of rules by following the same syntactical style and keeping the declarativeness of the rules as much as possible. For example, the problem of ordering of rules is solved by enumerating the rules explicitly. The interpreter is responsible for imposing this order. In that way the language remains declarative and is not extended with iteration and condition.

Metaprograms consist of rules called *metarules.* In addition, two variables are introduced and can be used within the rules. Metarules and ordinary transformation rules are given in the same transformation specification. Most metarules follow the style of the base level rules: they take a source element and execute instantiations, updates, and deletes.

In this section we present the language primitives for specifying metaprograms. We first explain the two variables, followed by a presentation of the metarules.

5.2.1 Variables

Two variables are introduced: *transformation* and *this*. The variable *transformation* may be used everywhere in the transformation specification. It refers to the transformation definition being executed. It is accessible as an ordinary model. OCL can be used for navigating over the transformation definition.

The variable *this* may be used in the context of a model element rule. During the execution it refers to the rule being executed. Via *this* a transformation definition may access the name of the rule, the source pattern, etc. Function *value* can be invoked on rules bound to *this* variable. It takes as argument an identifier that refers to the rule source or to the target elements and returns the element currently assigned to the identifiers in the context of the rule match.

For example, in the context of rule *allAttributesOfRoot* the expression *this.name* is evaluated to "allAttributesOfRoot". The property *name* is defined in the metamodel of MISTRAL [23]. The expression *this.value('class')* is evaluated to the value of the variable *class* for the concrete match of the rule. Variables *transformation* and *this* allow structural introspection. Function *value* allows introspection of runtime data that form the context of a rule execution. This is an example of behavioral introspection.

5.2.2 Metarules

Several types of metarules are introduced to allow behavioral reflection.

5.2.3 Transformation execution rule

This rule is derived on the basis of the analysis of the procedure *ExecuteTransformation*. It is called when the transformation execution starts and allows the programmer to alter the default transformation execution algorithm. More concretely, the programmer is able to specify a partial/complete order over model element rules and to modify the transformation definition before the execution. Thus, it provides reflection over the computational object of transformation definition and the computational event of executing the rules. The syntax is as follows:

```
ruleName TransformationExecutionRule
        target[actions]
        execute listOfRuleNames
        execute listOfRuleNames
        ..... .
        execute listOfRuleNames
```

The optional *target* construct lists actions that are executed over the transformation definition. Thus, a transformation definition may be changed before its execution. Actions are executed before the *execute* clauses. An *execute* clause lists at least one rule. Their execution order is determined by the transformation engine. *execute* clauses are ordered. This means that a rule included in a clause will be executed before the rules listed in the following clauses. In this way the programmer may specify a partial execution order, a full execution order (if only one rule is given in a clause) and may skip a rule by not including it in a clause.

Only one metarule of this kind is allowed per transformation. A rule may be referred to from at most one *execute* clause. If the list of *execute* clauses is empty then all the rules are executed in an order determined by the interpreter, that is, the default execution algorithm is used.

If a target element is requested and it is not created yet, the default execution algorithm will create the element on demand. This may conflict with the partial ordering of rules since the target element may be created by a rule that must be executed later. Such a request will result in a runtime error because the rule will not be matched and the information in the internal trace will be missing.

This rule may add new rules before the transformation execution, if needed, and modify existing rules. For example, adding the trace functionality can be done in a transformation execution rule. Section 6.1 shows an alternative solution.

5.2.4 Execution order rule

This rule provides behavioral reflection on the procedure *ExecuteRuleOnMatch*. It is invoked before the execution of a model element rule over a source element. This metarule controls the execution order of the rules that match the source element. Our motivation for introducing this metarule is the need for a fine-grained control over the rule order per single source element. This need is illustrated in the scenario about aspect weaving at shared join points. The syntax is as follows:

```
ruleName ExecutionOrderRule {
        source
        target[listOfRuleNames]
}
```

source has the same syntax as the source of model element rules. The target specifies an ordered list of rule names. If a source element is matched by the source of an execution metarule then during the execution of the transformation the set of rules that will be executed on this element is obtained from the metarule. Rules are executed in the order specified in *listOfRuleNames*. In general, this order is partial, that is, not all the applicable rules are listed there. Not listed rules may be executed in an order decided by the transformation engine.

5.2.5 Rule execution

This metarule provides behavioral reflection on the procedure *ExecuteRuleOnMatch*. Similar to the previous rule, it is invoked before the execution of a model element rule over a source element. The programmer is able to take actions, for example, to modify the rule or to create new elements in a model. The changes done on the rule are valid only for the execution in the given context. We will illustrate the applicability of this rule in Sect. 6. Its syntax is as follows.

```
ruleName RuleExecution on listOfRuleNames {
        source
        target
}
```

The syntax of *source* and *target* is the same as the one used in model element rules. The metarule is invoked only for rules listed in *listOfRuleNames* and whose match satisfies the *source* pattern. Thus, the programmer has fine-grain control over the execution context in which reflection is provided. Once the context is identified, the *target* is executed. After that the normal execution of the model element rule is resumed.

It is not possible to have more than one execution metarule for a pair of a model element rule and a source element. Multiple execution metarules may possibly conflict with each other by making conflicting modifications to the rule. The conflicts must be detected at runtime by the interpreter. We impose the aforementioned constraint to ease the implementation of the interpreter.

5.2.6 Instantiation rule

This metarule is called every time before executing an instantiation action. The syntax is the following:

```
ruleName InstantiationRule (inputParameters) {
        source
        target
}
```

The rule source matches elements in the source model. The metarule is executed only for those source elements. In that way the developer may narrow the scope of the rule. The rule accepts two input parameters: the first one is bound to the rule being executed and the second one is bound to the instantiation action being executed. These parameters give access to the context of the instantiation.

5.2.7 Slot assignment rule

This metarule is called every time before assigning a value to a slot in the context of a target action. The syntax is the following:

```
ruleName SlotAssignmentRule (inputParameters) {
        source
        target
}
```

The execution semantics of this rule is similar to the semantics of the instantiation rule. The difference is that three parameters are passed: the rule being executed, the identifier of the object that possesses the slot, and the slot name.

5.2.8 Resolution rule

This metarule is applied when a call to the internal resolution algorithm occurs. Recall that this algorithm is invoked by calling the predefined operation *trace*: *transformation.trace(element, id).* The syntax of the metarule is as follows:

```
ruleName ResolutionRule (inputParameters) {
        condition
        expresion
}
```

There are two input parameters: the source element and the target element identifier. An optional condition can limit the application of the rule to certain input values only. The expression specifies how the target element is obtained. The metarule can override the logic of the default resolution algorithm. For example, the developer may maintain their own trace structure and use it during transformation execution. Another example is that the resolution metarule may rename the target identifier and call the default execution algorithm with changed parameters. Calls to the default resolution algorithm from the metalevel may cause a cycle if a resolution rule intercepts a call. This is avoided by checking the source of the call. If it comes from the metalevel, the interpreter does not execute resolution rules and the computation is performed at the base level.

6 Example applications

We present three example applications of the reflective constructs introduced in MISTRAL. Section 6.1 presents a solution to the first motivating scenario. Sect. 6.2 presents a solution to the second scenario. Section 6.3 provides an additional example that illustrates the usage of instantiation and slot assignment metarules (not needed in the motivating scenarios).

The notation used in Sect. 3 to present the motivating examples is considered as pseudo code and only states the relations among the source and target elements. This section uses the MISTRAL syntax which is inspired by this pseudo notation.

6.1 Generation of trace links based on introspection

Section 3.1 presented a scenario in which the trace generation functionality is user defined. The solution given in a pseudo code exposes two problems. The first problem is the repetition of the trace generation code across rules. The second problem is the difficulty in adding and removing this functionality without manually editing the transformation definition.

We first present a solution in MISTRAL to the transformation described in Sect. 3.1.

```
1.  transformation UML2Java
2.  input s : UML
3.  output t : Java
4.
5.  copyPackage ModelElementRule {
6.    source [sp : UML!Package]
7.    target [tp : Java!Package {name = sp.name}]
8.  }
9.
10. copyClass ModelElementRule {
11.   source [sc : UML!Class]
12.   target [tc : Java!Class {name = sc.name, isPublic = true, isStatic = false,
13.               field = sc.attribute->collect(a | transformation.trace(a, 'f')) ,
14.               method = sc.attribute->collect(a |
15.                   Sequence{transformation.trace(a, 'getter'),
16.                            transformation.trace(a, 'setter')})->flatten()}]
17. }
18.
19. transformAttribute ModelElementRule {
20.   source [sa : UML!Attribute]
21.   target [f : Java!Field{name  = sa.name, isPublic = false,
22.                 type = transformation.trace(sa.type, 'tc'),
23.                 owner = transformation.trace(sa.owner, 'tc')},
24.           getter : Java!Method{name  = 'get_'+sa.name, isPublic = true,
25.                 returnType = f.type, owner = f.owner},
26.          setter : Java!Method{name  = 'set_'+sa.name, isPublic = true,
27.                 owner = f.owner}]
28. }
```

The following code is a solution for trace generation that is independent of the details of the rule in which it is executed. Such a generic solution may be obtained by using the variable *this*. The following instantiation named *trace* (line 8) needs to be added in every rule (we show only the new signature of the transformation and the first rule):

```
1.  transformation UML2Java
2.  input s : UML
3.  output t : Java, traces : Traces
4.
5.  copyPackage ModelElementRule {
6.    source [sp : UML!Package]
7.    target [tp : Java!Package {name = sp.name},
8.            trace : Traces!Trace in traces {ruleName = this.name,
9.               sourceName = this.value(this.source.variableName).name,
10.              targetName = this.target->select(t |
11.                           t.identifierName <> 'trace')->collect(t |
12.                           this.value(t.identifierName).name)}]
13. }
```

A new output model that will contain traces is declared in line 3. The new instantiation specified in lines 8–12 creates a new trace and puts it in the extent denoted with the variable *traces* (please note the keyword **in** on line 8). It should be noted that the new code does not use any details from the hosting rules. The concrete details about the elements are

obtained by the function *value*. The navigation expressions **this**.source.variableName and **this**.target rely on knowledge from the MISTRAL metamodel.

This solution, however, requires manual changes in every rule. We add a metarule that modifies model element rules by adding the trace generation functionality. In that way we achieve loose coupling between the existing rules and the trace code. This metarule exemplifies a structural intercession.

created and assigned to the variable *new*. Note that it is initialized with MISTRAL code. MISTRAL code can be used in initialization expressions in order to specify code fragments in the concrete syntax rather than in the more clumsy abstract syntax. The *update* action adds the trace instantiation *new* to the target of every rule. Here the variable *rule* refers to the model element rule controlled by the *addTrace* metarule.

```
1.  addTrace RuleExecution on copyPackage, copyClass {
2.    target [new : Mistral!Instantiation = trace : Traces!Trace in traces
3.               {ruleName = this.name,
4.                sourceName = this.value(this.source.variableName).name,
5.                targetName = this.target->select(t |
6.                                  t.identifierName <> 'trace')->collect(t |
7.                                     this.value(t.identifierName).name)}
8.            update rule{target = rule.target->append(new)}
9.           ]
10. }
```

We use a metarule that controls the execution of model element rules *copyPackage* and *copyClass*. Instead of using a list of names, the pattern symbol "*" can be used to select all the rules in a given transformation. A new instantiation is

6.2 Controlling aspect weaving at shared join points

The solution to the second scenario (see Sect. 3.2) is given below.

```
1.  addLogConcern ModelElementRule {
2.    source[sourceMethod : Java!Method condition{(sourceMethod.name = 'methodA')
3.                                        or (sourceMethod.name = 'methodC')}]
4.    target[logInvocation : Java!MethodCall{variableName='Log',
5.                                           methodName = 'log'},
6.           update sourceMethod{statements =
7.             Sequence{logInvocation}->union(sourceMethod.statements)}
8.          ]
9.  }
10.
11. addSynchConcern ModelElementRule {
12.   source[sourceMethod : Java!Method]
13.   target[obtainLockInvocation : Java!MethodCall{variableName = 'Lock',
14.                                                 methodName = 'getLock'},
15.          releaseLockInvocation : Java!MethodCall{variableName = 'Lock',
16.                                                  methodName = 'releaseLock'},
17.          update sourceMethod{statements =
18.         Sequence{obtainLockInvocation}->union(sourceMethod.statements)->union(
19.                                           Sequence{releaseLockInvocation})}
20.         ]
21. }
```

As can be seen, the logging aspect is applied to methods with name *methodA* or *methodC*. The synchronization aspect is applied to all the methods in the base program. These two aspects have shared join points. Assume that for some methods we want to apply first the logging aspect and then the synchronization aspect, while for other methods we reverse the order. The execution order metarule can be applied in this situation. It allows to select a source element and to specify a partial order of execution among the rules that match the element. We apply the following execution order rule:

```
orderConcerns ExecutionRule {
  source [sourceMethod : Java!Method condition{sourceMethod.name = 'methodA'}]
  target [addSynchConcern, addLogConcern]
}
```

The rule specifies that for all methods with name *methodA* first the synchronization aspect is applied and then the logging aspect. If there are other rules matching that method then their order is up to the execution engine. The execution order metarule implements behavioral intercession in MISTRAL.

6.3 Generation of execution trace

A declarative language like MISTRAL usually does not rely on explicit specification of the control flow. The execution engine detects dependencies among rules at runtime and orders the execution of instantiations and slot assignments. Typically, the execution order and the dependencies are not kept after the execution. Their externalization, however, may help in solving a number of problems. In this example we show how the execution trace can be captured by intercepting execution events. Instantiation rules and slot assignment rules are used for this purpose.

Consider the previous example of transforming UML to Java. We introduce two metarules: one for intercepting instantiations and one for intercepting slot assignments.

```
1. catchInstantiation InstantiationRule (rule : Mistral!Rule,
2.                                       instantiation : Mistral!Instantiation) {
3.   source[s : SimpleUML!UMLModelElement]
4.   target [e : ExecutionEvents!InstantiationEvent in events{ruleName= rule.name,
5.                    sourceName = s.name,
6.                    id = instantiation.identifierName,
7.                    targetClassName = instantiation.type.elementName}]
8. }
9.
10. catchAssignment SlotAssignmentRule (rule : Mistral!Rule,
11.                                     id : Mistral!String,
12.                                     assignment : Mistral!SlotAssignment) {
13.  source[s : SimpleUML!UMLModelElement]
14.  target [e : ExecutionEvents!PropertyAssignment in events {sourceName= s.name,
15.                    ruleName = rule.name,
16.                    id = id,
17.                    slotName = assignment.name}]
18. }
```

The*catchInstantiation* rule is invoked before executing an instantiation from a given rule on a given source node. The source node is bound to the source variable *s*. The context rule and the instantiation are passed as parameters. The source of the rule may have a condition, thus limiting the scope of the rule only on some source nodes. When the rule is executed a new instance of the class *InstantiationEvent* will be created. It will store info about the name of the rule, the name of the source, the identifier of the target element, and the target class name. Clearly, this is just one possibility to store information about instantiation events.

Similarly, *catchAssignment* rule is invoked every time before an assignment is performed. Again, the source node is passed to the rule and the context of the execution is passed as three parameters.

We envision two applications of these metarule types. They may be used for performing debugging of transformations. They may also be used to capture the execution order and the dependency among the execution events. This information, combined with information about the access to the source and target model elements and a trace record (like in the previous example), allows performing change impact analysis and to manage change propagation when the source model is changed. We intend to report about this application of reflection in another paper.

7 Discussion

In our experience with reflection we encountered several challenges that posed limitations and required making decisions among possible solutions. They are discussed in the following subsections.

7.1 Identification of reflective features

Before implementing a reflective framework the first step is the identification of the elements to be exposed to the metalevel. In our approach we started with a set of scenarios with problems and used them as a criterion for selecting among the possible reflective features. This approach, however, may not be sufficient if a general-purpose reflective mechanism is required. To overcome this, we extended our analysis by inspecting the computational objects and events available in the execution semantics of the language.

We cannot claim that we support full reflection according to the view expressed in [21], where every computational object has its own metaobject. First, some aspects of the computation were intentionally left out. A general principle in language design is that a good language protects its own abstractions. In contrary, reflection requires that some, otherwise implicit, computational aspects are exposed. We decided not to expose the rule matching algorithm and the internal trace. Second, MISTRAL is a work in progress. Several features are planned but not fully implemented yet. They are not considered in the paper. These are rule inheritance, constructs for transformation packaging and reuse, and helper rules. A detailed presentation of these features is given in [16]. Rule inheritance and package import are resolved statically before the execution of the transformation. We do not expect conceptual problems in including these language constructs in the reflective mechanism by providing an interceptor to a suitable event and an access to the transformation model. The same is valid for the helper rules due to their close similarity to the model element rules. It should be noted that the algorithm of resolution of the rule inheritance is an interesting computational event that can be exposed to metaprograms. This would allow transformation developers to alter the default inheritance semantics for rules.

Some metarules required more expressivity from the language. However, we perceive the domain-specific and declarative nature of MISTRAL as a higher priority than the richness of its reflection capabilities. This leads to limitations discussed in the next subsection.

7.2 The reflective tower

It is known that the computation levels may be layered upon each other forming a potentially infinite stack of computation levels known as a *reflective tower*. The metalevel may be perceived as an ordinary computation (base) level. Another metalevel may be built upon it. Thus, the dichotomy base/meta level becomes relative. In MISTRAL, we use only two computation levels. This means that it is not possible to specify metaprograms upon other metaprograms. This is certainly a major limitation of our approach.

There are benefits in using more than two levels. One of the applications of reflection is debugging. However, if we want to debug a transformation for which metarules are specified we need a reflection upon the metarules. Another reason to consider more than two levels is that metarules are often modules with reusable application logic. They may be manipulated in the same way as a metarule manipulates base rules.

We already mentioned the problem that certain computations cannot be easily expressed by using base rules of MISTRAL. The reason is the domain-specific nature of the language that makes it suitable for a limited set of problems. If we decide to build a second (or a higher order) metalevel, the same challenge will appear: we have to analyze the required scenarios and eventually to extend further the language. This can be stopped if we introduce constructs from general-purpose languages such as iteration and condition. We consider the domain-specificity of the language as a higher priority.

Metaprogram can be expressed in a general-purpose programming language. If this language is reflective then the

problem with the reflective tower is solved according to the existing research in this area [32]. The reason not to choose this approach is that we wanted to apply a solution within the boundary of MISTRAL, that is, a solution which is still domain-specific.

An interesting direction for research is to study a transformation language implemented as an embedded DSL [9] in a language that supports reflection. An example is the RubyTL [5] implemented in Ruby. This idea is a subject of future work.

7.3 Implementing reflection

In general, there are three ways to introduce reflection in a language: using preprocessing, modifying the language interpreter, and modifying the language compiler.

- *Using preprocessing.* This approach is employed in [19]. The language is syntactically extended and the new constructs are translated to the existing constructs by including a preprocessing phase. This approach does not require changing the interpreter/compiler of the language. In the context of MDE, this approach may be applied by using a higher-order transformation that translates the reflective program to a non-reflective one. Jouault [11] applies HOT to achieve flexible traceability. He does not extend the transformation language with new features though. The shortcoming of the preprocessing approach lies in the fact that it deals with static aspects of reflection and is limited to source code manipulation. In case of reflection upon the runtime structures, HOTs are not enough.
- *Changing the interpreter.* The current prototype of the MISTRAL engine is implemented as an interpreter and we had to change it to introduce the forms of behavioral reflection we presented. Generally, this approach is considered to lead to decreased performance in terms of execution time. This is due to the fact that the interpreter is checking every time if a reflection feature is requested when the interpreter evaluates expressions that are exposed to metacomputations.
- *Changing the compiler*. This approach overcomes the problems of the interpreter-based execution. During compilation the compiler may analyze the reflective code and introduce the invocations to the metalevel only when necessary. An application of this idea is reported in [28] and applied for the Java reflective framework Reflex.

8 Related work

Unfortunately, there is limited experience in using reflection in current model transformation languages. The most commonly found form of reflection is structural introspection over model elements based on the reflective API of ECore.

Tefkat [18] indicates reflection support. This comes in three forms. The first form is a generic access to the properties and metaclasses of the model elements. Tefkat relies on the ECore reflective API to do that (the operational environment of the language is based on the ECore metamodeling architecture). The second form allows specifying expressions in places where a class or a feature is expected. The third form uses the construct *AnyType* that allows any object to be selected regardless of its concrete type. The latter two mechanisms may not be regarded a reflective features but they increase the genericity of the Tefkat programs.

These three reflective constructs allow *model copy* transformations to be specified in a generic and concise manner. The reflective support in Tefkat concerns only source and target model elements. We are not aware of reflective capabilities that allow navigation over the transformation rules or changing the behavior of the transformation system at runtime. MISTRAL does not support introspection over model elements in the form found in Tefkat.

VIATRA2 [31] allows defining generic template rules. In these rules, the classes are parameters that may be substituted via template instantiation. The instantiation is achieved by a meta-transformation in the terms of VIATRA2. In fact, the instantiation executes a higher-order transformation which manipulates the generic transformation.

MISTRAL does not directly allow generic template rules as first-class language elements. This effect can be achieved by using transformation and rule execution metarules that replace the parameters with concrete classes.

We have not experimented with transformation languages that are embedded DSLs in a language with reflective features. Such a language is RubyTL [5] implemented in the context of Ruby. The authors of RubyTL report on the possibility to introspect the transformation program elements, a capability generally available for Ruby programs. This corresponds to the capability provided by the variables *transformation* and *this* in our work.

Our work fits in the more general context of adding reflection to DSLs. The closest related work in this domain comes from the series of workshops on Domain-Specific Aspect Languages (DSAL) [6]. One of the problems treated in this research area is how to specify domain-specific join points in a DSL and how to implement domain-specific aspect weavers. Reflection is perceived as a solution to this problem.

The reflective features of MISTRAL allow modularization of crosscutting transformation code. The metarule *addTrace* in Sect. 6.1 can be perceived as an aspect specification. The scattered trace generation code is modularized in a single rule. This rule is executed on two rules. This corresponds to join points in the AOP terminology. The *update* action in the

rule implements the advice mechanism that inserts the aspect code in the selected base rules.

9 Conclusions

In this paper we studied the possibilities to employ reflection in a rule-based model transformation language. The design of the reflective framework was derived from several usage scenarios and the computational objects used in the transformation execution. The reflective features were considered from a more general perspective in a two-dimensional space that provides a reasoning framework about the possible solutions. The reflective capabilities were implemented in an experimental model transformation language by modifying the language interpreter.

It was possible to solve the problems formulated in the scenarios. The trace generation may be separated in a single module, either in a transformation execution metarule or in a rule execution metarule. The metarules may be excluded if this functionality is not needed. A fine-grained weaving policy in the second scenario may be expressed via a global rule ordering (transformation execution rule) or via an ordering per single element as shown in Sect. 6.2.

Since we applied reflection on a DSL we encountered several limitations rooted in the fact that a DSL has a limited problem scope and often limited expressivity. This prevented us to achieve uniformity in treating base rules and metarules. The result is a reflective architecture with a single metalevel.

The major benefit of using reflection is achieving transformation solutions with better quality. We were able to specify generic and reusable trace generation functionality. This should help us in improving traceability and change management in model transformations. Reflection may provide fine control during execution and as we suggested in the paper other technologies (e.g., AOP) may benefit from this.

References

1. Akehurst, D., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra — Simple Transformations in Java. MoDELS 2006, pp. 351–164. Genova, Italy (2006)
2. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. ECOOP Workshop on Object-based Distributed Programming, pp. 152–184. Darmstadt, Germany (1993)
3. Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S.A., Grose, T.J.: Eclipse Modeling Framework. Addison Wesley, Reading (2003)
4. Chiba, S.: Load-time structural reflection in Java. ECOOP 2000, pp. 313–336. Sophia Antipolis, France (2000)
5. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A Practical, Extensible Transformation Language. ECMDA-FA 2006, pp. 158–172. Bilbao, Spain (2006)
6. Domain-Specific Aspect Languages Workshops Series. http://dsal.dcc.uchile.cl
7. Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: Detecting and Resolving Ambiguities Caused by Inter-dependent Introductions. AOSD 2006, pp. 214–225. Bonn, Germany (2006)
8. Herzeel, C., Costanza, P., D'Hondt, T.: Reflection for the Masses. Self-Sustaining Systems 2008, pp. 87–122. Potsdam, Germany (2008)
9. Hudak, P.: Building Domain-Specific Embedded Languages. ACM Comput. Surv **28**(4es), 196 (1996)
10. Jouault, F., Kurtev, I.: Transforming Models with ATL. Model Transformations in Practice Workshop, MoDELS 2005 Conference, Montego Bay, Jamaica (2005)
11. Jouault, F.: Loosely Coupled Traceability for ATL, ECMDA 2005 Workshop on Traceability. Nuremberg, Germany (2005)
12. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. FMOODS 2006, pp. 171–185. Bologna, Italy (2006)
13. Kiczales, G., Rivières, J.D., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)
14. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. EOOP 2001, pp. 327–354. Budapest, Hungary (2001)
15. Kojarski, S., Lorenz, D. H.: Awesome: an Aspect Co-weaving System for Composing Multiple Aspect-Oriented Extensions. OOPSLA 2007, pp. 515–534. Montreal, Canada (2007)
16. Kurtev, I.: Adaptability of Model Transformations, PhD thesis, University of Twente, The Netherlands. ISBN 90-365-2184-X (2005)
17. Kurtev, I., van den Berg, K., Jouault, F.: Rule-based modularization in model transformation languages illustrated with ATL. Sci. Comput. Program **68**(3), 138–154 (2007)
18. Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. MoDELS Satellite Events. pp. 139–150, Montego Bay, Jamaica (2005)
19. Leitner, A., Eugster, P., Oriol, M., Ciupa, I.: Reflecting on an Existing Programming Language. TOOLS Europe 2007, JOT **6**, (9) (2007)
20. Lewis, B., Ducassé, M.: Using events to debug Java programs backwards in time. OOPSLA Companion 2003, pp. 96–97. Anaheim, CA, USA (2003)
21. Maes, P.: Computional reflection. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium (1987)
22. McAffer, J.: Engineering the meta-level. Reflection 1996, pp. 39–61. San Francisco, CA, USA (1996)
23. Mistral web site. http://www.vf.utwente.nl/~kurtev/mistral/
24. OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10 (2002)
25. Paakki, J.: Attribute Grammar Paradigms—A High-Level Methodology in Language Implementation. ACM Comput. Surv **27**(2), 196–255 (1995)
26. Pawlak, R., Seinturier, L., Duchien, L., Floring, G.: In: JAC: A flexible solution for aspect-oriented programming in Java. 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns, Japan (2001)

27. Smith, B.C.: Reflection and Semantics in Lisp. In: 14th Annual ACM Symposium on Principles of Programming Languages 1984, pp. 23–35. Salt lake City, USA (1984)
28. Tanter, E., Noyé, J., Caromel, D., Cointe, P.: Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. OOPSLA 2003, pp. 27–46. Anaheim, CA, USA (2003)
29. Tanter, E.: From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming. PhD thesis, University of Nantes and University of Chile (2004)
30. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bezivin, J.: On the Use of Higher-Order Model Transformations. ECMDA 2009, pp. 18–32, Enschede, the Netherlands (2009)
31. Varro, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. UML2004, pp. 290–304. Lisbon, Portugal (2004)
32. Wand, M., Friedman, D.P.: The mystery of the tower revealed: a non-reflective description of the reflective tower. In: ACM Symposium on LISP and Functional Programming 1986, pp. 298–307. Cambridge (1986)

Author Biography

Ivan Kurtev holds a MSc degree in Computer Science from University of Sofia and a PhD degree in Software Engineering from University of Twente. He is currently an assistant professor in the Software Engineering group in University of Twente, the Netherlands. His main research interests are in the domain of Model Driven Engineering with a focus on model transformation languages, metamodeling, requirements modeling and traceability.