SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **00**:1–29
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Scheduling model-to-model transformations with continuations

Jesús Sánchez Cuadrado*[1] and Jesús Perera Aracil[2]

[1] *Universidad Autónoma de Madrid*
[2] *Universidad de Murcia*

SUMMARY

Model transformations are at the heart of Model Driven Engineering (MDE) since they allow the automation of diverse kinds of model manipulations. Transformation scheduling is a key issue in the design and implementation of many transformation languages. This paper reports our results using continuations as the underlying technique for building a scheduling mechanism implicitly driven by data dependencies among transformation rules. In order to support our experiments, we have built a proof-of-concept model transformation language which is also reported here. First, we motivate the problem by analyzing the scheduling mechanism of current model transformation languages. Then, we introduce the notion of continuation, showing its applicability to model transformations. Afterwards, we present our approach, notably explaining how dependencies are specified and giving the scheduling algorithm. We also analyze lazy resolution of rules, and how to deal with collection operations. The approach is validated by an implementation that targets the Java Virtual Machine, and running performance benchmarks which show its efficiency and scalability. Besides, we discuss how it can be applied to other existing transformation languages, and present several applicability scenarios. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Model-to-model transformation; Transformation scheduling; Transformation algorithm

1. INTRODUCTION

Model transformations are at the heart of Model Driven Engineering (MDE), since they allow the automation of diverse kinds of model manipulations. Examples include mapping and synchronization of models, reverse engineering, model refactoring, code generation, etc. Particularly, model transformations that take a model conforming to some meta-model and generate a different model, possibly conforming to a different meta-model are called model-to-model transformations [1]. Several model-to-model transformation languages with different features have been proposed over the last years. As described in [2], an important feature is transformation scheduling, which determines the way in which transformation rules are executed. A design option is to choose an *implicit scheduling mechanism*, so that the execution order of the transformation rules is computed according to their dependencies (i.e., there are no dedicated constructs to set the execution order explicitly). Well-known examples of languages implementing this feature are ATL [3], Relational QVT [4] and Tefkat [5]. Hence, devising flexible and efficient rule scheduling mechanisms is a key issue in the design and implementation of these kinds of languages.

*Correspondence to: Ciudad Universitaria de Cantoblanco, 28049. Escuela Politécnica Superior. E-mail: jesus.sanchez.cuadrado@uam.es

Contract/grant sponsor: Spanish Ministry of Economy and Competitivity and R&D programme of Madrid Region; contract/grant number: TIN2011-24139 and S2009/TIC-1650

On the other hand, the concept of *continuation* has proved useful in a variety of areas such as compiler construction [6], web servers [7], concurrency [8], etc. A continuation reifies the concept of “the rest of the computation”, so that the execution state of a given program can be saved into a continuation and restored later. This concept is supported in different forms in some programming languages, for instance Scheme [9], Ruby [10] or Scala [8]. We believe that this characteristic can be used to devise flexible and efficient strategies for rule scheduling in model-to-model transformation languages.

In this paper we present a novel scheduling mechanism for model-to-model transformations which is based on continuations. The underlying idea is to describe data dependencies among transformation rules by setting constraints on the trace model (i.e., a trace model holds correspondences among transformed elements). Whenever a rule requires a target element that is not readily available through the trace model because it has not been created yet, it saves its execution state into a continuation, which is resumed later when the element is available (i.e., it is provided by some other rule). Meanwhile, other non-dependent parts of the transformation are executed. Besides this basic behaviour, there are other issues that must be addressed, namely: supporting lazy execution of rules, tackling stuckness (i.e., when not all elements can be resolved), and dealing with collection operations combined with continuations. In order to support our experiments, we have built a proof-of-concept transformation language, called Koan, which will be used to illustrate the concepts in the paper. Nevertheless, as will be explained, our approach is not specific to Koan but can be used to implement the scheduling mechanism of other transformation languages.

We have validated our approach by implementing Koan on top of the Java Virtual Machine (JVM) and EMF/Ecore [11], testing it by developing several model transformations. In particular, this paper will be illustrated with an excerpt of a large reverse engineering model transformation taken from the MoDisco project [12] that we have re-implemented in our language. Moreover, we have performed several performance benchmarks which show that our approach based on continuations is efficient and scalable. Finally, the increased level of flexibility, that our continuation-based approach provides, have proven useful to deal with advanced transformation scenarios such as language interoperability and transformation composition [13].

Paper organization. Section 2 analyzes the scheduling mechanism of current model transformation languages in order to motivate the need of new, more flexible scheduling mechanisms. Section 2 gives some background about continuations and motivates its usage in model transformations. Section 4 introduces Koan through an example, then focusing on how to describe dependencies and tackling lazy rule execution, while Section 5 explains the underlying execution algorithm. Section 6 presents the related work and explains how our approach could be used in other transformation languages. Section 7 introduces some applicability scenarios of our approach and compares the performance of Koan against other languages. Finally, Section 8 concludes and presents the future work.

2. MOTIVATION

This section introduces model-to-model transformations (M2M), in particular focusing on the scheduling part of different approaches to model transformation. Additionally, the running example that will be used throughout the paper is presented.

2.1. Model transformations

Model transformations play a key role in Model-Driven Engineering as they enable the automatic manipulation of models. A transformation is the automatic generation of a target model from a source model, according to a given transformation definition [14]. Model transformations can be classified, among other criteria [1], into exogenous or endogenous. In an endogenous transformation both the input and output meta-models are the same, whereas in an exogenous transformation both meta-models are different. For instance, a model refactoring is an example of endogenous transformation, and a translation from one language to another is an example of exogenous transformation. Roughly, if a transformation takes only one model (acting both as input and output model) it is called in-place, whereas if it takes an input model and creates an output model it is called model-to-model.

Model transformations are usually developed using specialized languages, called model transformation languages. Each language typically provides a set of features that makes it more suitable to target a type of transformation. Endogenous transformations are more naturally implemented using in-place model transformation languages, whereas exogenous transformations are normally implemented using model-to-model transformation languages (M2M languages) because they provide specialized constructs, but any combination is possible in practice.

As a running example, we will use a reverse engineering transformation that takes a model conforming to the abstract syntax of Java (represented as a meta-model) as input, and it outputs a model conforming to the Knowledge Discovery Metamodel (KDM), which is a standard metamodel proposed by the OMG to represent software systems in software modernization [15]. This example is inspired by a similar transformation that is part of the MoDisco tool [12], but implemented in ATL. It is therefore an exogenous model transformation implemented with a M2M language.

Relevant excerpts of the meta-models for Java and KDM are shown in Figure 1(a) and Figure 1(b) respectively. In the Java meta-model a class is represented with a ClassDcl, which is composed of other declarations, such as MethodDcl that has a return type. The UnresolvedTypeDcl metaclass represents those classes that are referenced in the code but MoDisco could not resolve. The concepts in KDM are similar, but are expressed in a more general way. Thus, a class is represented as a ClassUnit composed of “code items” (codeElements reference), being MethodUnit one of those. A method has a Signature (connected through the type reference) that provides the name of the method, its parameters, and its return type, represented as a ParameterUnit with kind equals to ”return”.

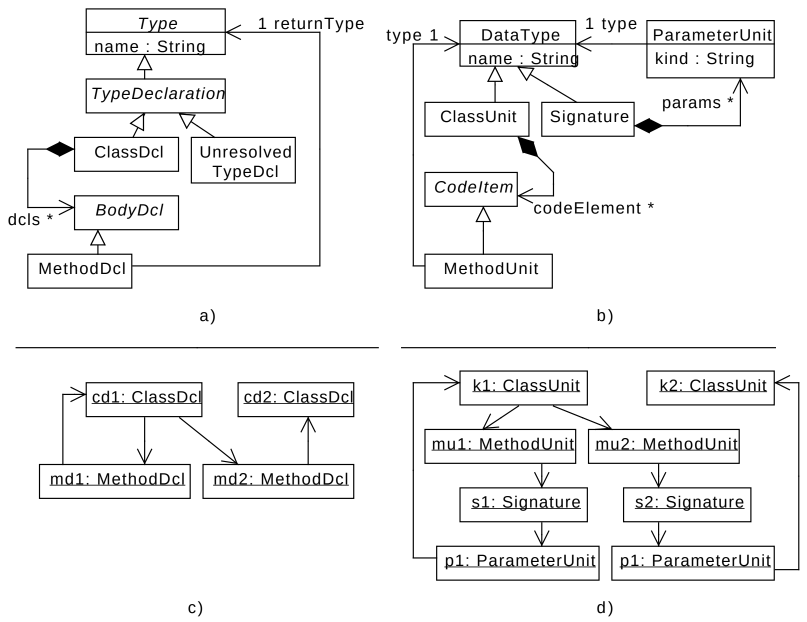


Figure 1. Metamodels excerpts for the Java2Kdm running example. (a) Java abstract syntax metamodel. (b) KDM metamodel. Below a simple Java model (c) and its KDM counterpart are shown (d).

Addressing this small transformation with a given model transformation language involves devising a solution in terms of the available constructs of the selected language. In general, the following four aspects are inherently present in the case of the example:

1. The one-to-one mapping between the Java ClassDcl and KDM ClassUnit metaclasses
2. The one-to-n mapping between the Java MethodDcl and KDM MethodUnit, Signature and ParameterUnit metaclasses.

3. Filling the codeElement reference of ClassUnit with the corresponding KDM MethodUnits, that have to be retrieved from their counterparts in the source model, MethodDcl.
4. Setting the type reference of ParameterUnit with the corresponding KDM ClassUnits, that have to be retrieved from their counterparts in the source model, ClassDcl.

As can be observed, in this example, issues 1 and 2 are more or less straightforward (i.e., pattern matching is very simple), whereas the main difficulties arise in issues 3 and 4. First, a mechanism to retrieve a target element from a source one has to available. Secondly, there may be circular dependencies: a class may refer to a method that refers to the same class directly or indirectly (in the source model of Figure 1, the `cd1` class refers to the `md1` method that in turn refers to `cd1`). Tackling these issues requires two kinds of mechanisms [2]:

- *Tracing*. It allows source-target relationships to be stored, so that, given a model element their correspondences can be found. There are two variants: implicit and explicit tracing, depending on whether the transformation engine implicitly creates the trace elements as rules are executed or if the user is in charge of this task.
- *Scheduling*. It is in charge of determining the rule execution order so that data dependencies among rules can be satisfied. There are two variants: implicit and explicit scheduling. In the first case, the transformation engine implements some algorithm that selects the execution order, whereas in the latter case the user is in charge of selecting the execution order using some dedicated construct of the transformation language.

Exogenous transformations are more naturally developed with M2M languages because their tracing and scheduling mechanisms are specifically designed to address this type of transformation. Although this is the main focus of this work, in-place transformations can also be used for this task since they provide general model manipulation mechanisms.

In the following, several model transformation approaches are briefly introduced, illustrated with the running example, and showing how the scheduling aspect is addressed. First, in-place transformation languages are discussed, then model-to-model transformation languages, and finally an assessment of the advantages and disadvantages of each approach is given.

2.2. *In-place model transformation languages*

In-place model transformation languages are normally implemented using graph rewriting techniques. Instances of tools for graph rewriting are VIATRA [16], AGG [17], Henshin [18] and ATOM^3 [19]. Triple Graph Gramars [20], while graph-based, are a model-to-model approach as they are specifically designed for exogenous transformations.

In general, developing an exogenous transformation using a language based on graph rewriting involves the following steps:

1. Creating a type graph that contains the source meta-model, the target meta-model and a set of intermediate types that relates metaclasses of both meta-models (i.e., a trace meta-model).
2. Creating transformation rules with the form $LHS + NACs => RHS$ (Left-Hand-Side, Non-Application-Condition and Right-Hand-Side). The LHS is the subgraph to be matched, the NAC is a subgraph that prevents rule application if matched, and the RHS is the new status of the subgraph when the rule is applied. New elements with respect to the LHS are created, and matched elements not appearing in the RHS are deleted.
3. Taking into account the tracing aspect of the transformation. The RHS of the transformation rule must explicitly create the trace links relating source and target elements.
4. Controlling the rule execution order. Although it is not a a compulsory feature, practical graph transformation languages normally provide some explicit scheduling mechanism to establish the order in which rules are executed. Otherwise, it is normally necessary to design the transformation rules in a certain way and using NACs to deal with circular dependencies and with trace links.

Figure 2 illustrates these steps assuming that rules are selected non-deterministically, and executed as long as possible. Rule (1) matches single ClassDcl elements and creates the corresponding ClassUnits. Interestingly, the NAC condition establishes that the rule must not be applied if a trace element has already been created for the source element. This is needed because rules are applied as long as possible. Rule (2) maps a MethodDcl to a MethodUnit, a Signature and a Parameter. However, to set the type reference of Parameter it needs to obtain the ClassUnit that corresponds to the ClassDcl, that is, the returnType of the matched MethodDcl. To this end, the LHS also matches the C2C trace link. Rule (3) establishes the link between the target KDM classes and methods, using a similar strategy as the previous rule: matching the trace model in the LHS. Actually, rule 3 could be defined together with rule 2 if the graph transformation tool supports non-injective pattern matching.

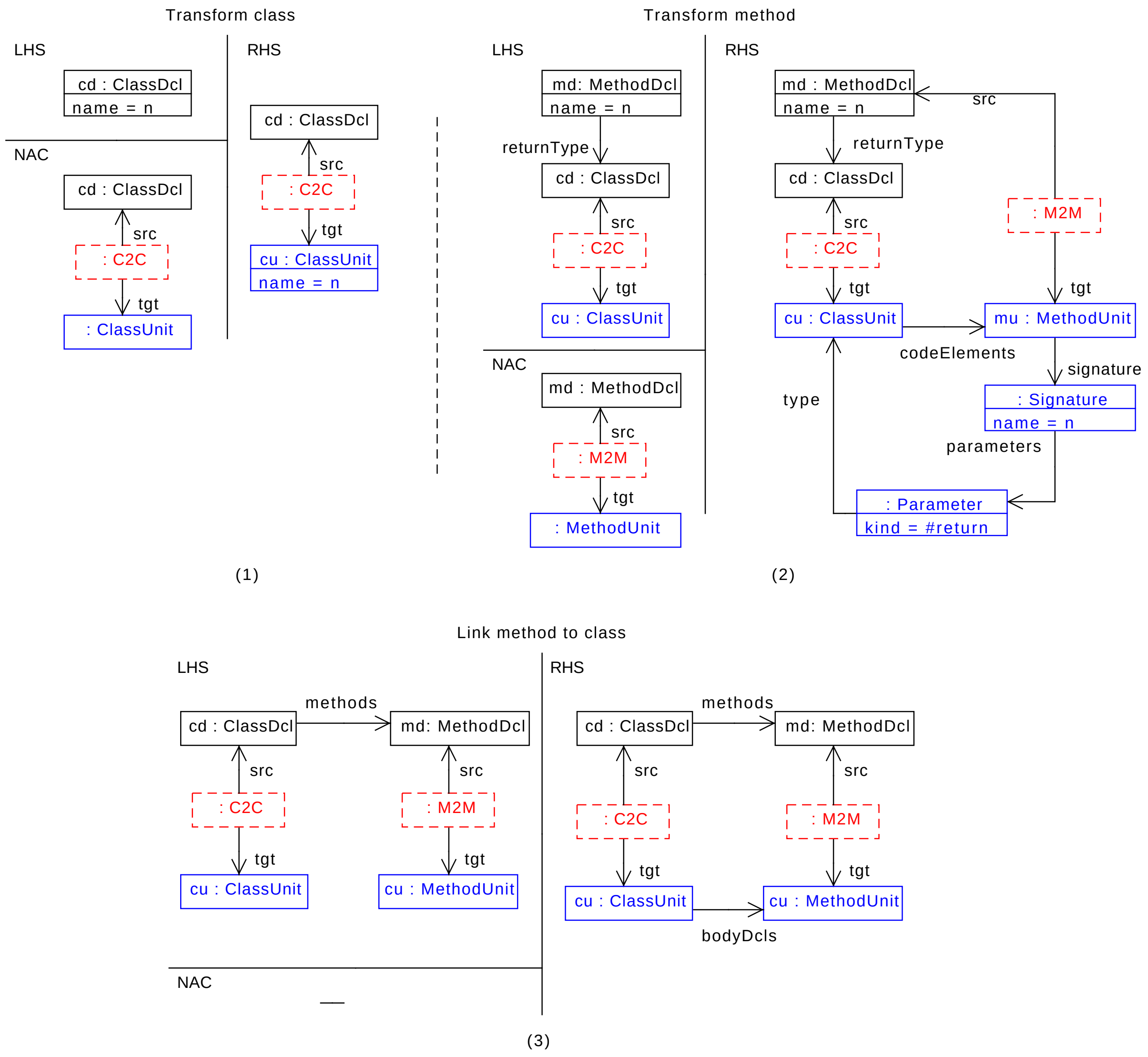


Figure 2. Running example expressed with a graph transformation approach. Black boxes correspond to source elements, blue boxes correspond to target elements, and red, dashed correspond to trace links.

As can be observed, the tracing mechanism is explicit since trace links are created in the RHS of the rules, and matched in the LHS as a way to retrieve the correspondences. Scheduling in this case is mostly implicit since no dedicated construct to control the rule execution order is used, but the transformation engine would apply rules whose LHS is matched (and no NAC is matched) until no rules are applicable. Hence, it is needed to define NACs with the only purpose of avoiding the same rule to be scheduled for execution over and over again. Also, the linking of the methods has been moved to rule 3 (unless non-injective matching is available). Thus, it is important to note that the scheduling has to be guided using NACs and by properly designing the rules. This scheduling is, to some extent, specific to the transformation problem being addressed: an exogenous transformation.

In this way, graph-based transformation languages normally provide explicit scheduling mechanisms in order to avoid the need of encoding scheduling aspects into the transformation rules. Next, four representative scheduling mechanism are briefly commented:

- *Non-deterministic choice* (implicit scheduling). This is the basic mechanism, presented in the example. It typically requires some care when designing the rules, to avoid conflicts and infinite recursion.
- *Layers* (explicit scheduling). This mechanism allows a transformation to be organized into sets of rules (a layer) that are executed together using non-deterministic choice [17]. Layers are executed in sequence, so that rules in a layer do not cause conflicts with rules in the other layers. In the example, rules could be organized in three layers with one rule each (1, 2 and 3). This could improve performance, but NACs to avoid rule re-matching will still be needed if rules are executed as long as possible in each layer. A similar approach is the use of priorities [19], in which rules with the highest priority are executed first, so that rules with different priorities do not conflict.
- *Abstract State Machines (ASMs)* (explicit scheduling). It is a well known formalism to specify the expected behaviour of a system [21]. It has been used in the VIATRA2 framework to describe imperative control structures in order to assemble graph transformation rules.
  In a language with explicit scheduling, such as VIATRA2, the use of NACs and the matching of trace links are not always required if the rules are properly scheduled using the ASM-based control language.
- *Story Diagrams* (explicit scheduling). It is visual notation based on UML activity diagrams [22] to describe the control flow of a graph transformation. The activities of the diagram embed graph rewrite rules, which can be parameterized with input/output data that is propagated along the transitions.

In general, it can be said that using a graph-rewriting transformation language to deal with exogenous transformation requires some explicit rule scheduling, either with explicit constructs or by encoding it in the rules.

2.3. *Model-to-model transformation languages*

The inherent complexity of exogenous transformations has been explained by going through the steps required when using an in-place approach. This provides a foundation to understand what is provided by M2M languages, which are specialized in these kinds of transformations. In essence, a M2M language embeds into its engine the logic to deal with trace information and facilitates rule scheduling, normally by making it implicit.

There are several model-to-model transformation languages in use today, such as QVT [4], ETL [23], Tefkat [5] and RubyTL [24], but ATL [25] is probably the most widely used one. Also, Triple Graph Grammars (TGGs) is a graph-based approach to (bidirectional) model-to-model transformations.

In the following, the style of M2M languages will be illustrated by using ATL. Afterwards, other approaches will be briefly described. An excerpt of the Java2KDM example written in ATL is shown in Listing 1.

```
rule transformClass {
  from cd : java!ClassDcl
  to    cu : kdm!ClassUnit (
    name <- cd.name,
    codeElements <- classDcl.methods
  )
}

rule transformMethod {
  from jMethod    : java!MethodDcl
11    to  methodUnit :  kdm!MethodUnit (
12      type <– signature
13    ),
14    signature  :  kdm!Signature (
15      name <– jMethod.name
16      parameters <– returnParameterUnit
17    ),
18    returnParameterUnit : kdm!ParameterUnit (
19      kind <– #return
20      type <– jMethod.returnType
21    )
22  }
```

Listing 1: Example transformation with ATL

The transformClass rule maps every Java class (ClassDcl) to a KDM ClassUnit (lines 1-7), while the transformMethod rule maps every Java method to a MethodUnit, but also creates a Signature element and a ParameterUnit element to represent the returned element (for simplicity, we assume that even methods without return value have a ParameterUnit with no type). ATL uses the concept of *binding* to specify the relationship between a source property and a target property. There are primitive bindings, that just assign primitive values, and non-primitive bindings that may require resolving a target element from a source element. For instance, line 4 is a primitive binding assigning the name of the source class to the target element's name, lines 12 and 16 are also straightforward as they just assign a target element created as part of the same rule. However, line 5 does require a resolution mechanism to obtain, for each source method, the target counterpart that has been generated by the transformMethod rule. Similarly, line 20 must resolve a ClassDcl element (that has been obtained through the jMethod.returnType expression) to the corresponding ClassUnit.

As can be observed in the transformation definition, the rule execution order is not given explicitly (e.g., with explicit calls). Instead, the ATL transformation engine schedules rule execution internally so that dependencies (ATL bindings in this case) can be resolved. This is another example of *implicit scheduling*.

In this way, some algorithm to establish a rule execution order that ensures a proper resolution of elements is required. For illustrative purposes, we briefly introduce the algorithm for ATL. The pseudo-code shown in Listing 2 is a simplification of the algorithm reported in [26]. For the sake of simplicity, we assume that only one target element is created per rule, and that the right part of a binding is single-valued (i.e., it is not a collection), but it can be easily generalised.

```
# First  phase: matching and storing trace links
ForEach rule R {
  ForEach element S that matches R.from {
    create target element T according to R.to
    create TraceLink for  R, S and T
  end
end

# Second phase: rule application
ForEach TraceLink L {
  R = the rule associated to L
  S = the matched source element of L
  T = the created target element of L

  # A binding B has the form T.feature  = expression in the context of S
  ForEach binding B declared for R {
    expression   =   initialization   expression of B
    value          = evaluate expression in the context of S
     if value needs resolving
       find trace link L2 where L2.S = value
      set T.feature to L2.T
    else
      set T.feature to value
    end
  end
```

```
end
```

Listing 2: Execution algorithm of the declarative part of ATL

The algorithm makes use of trace links to keep correspondences between source and target elements, in order to allow resolving a target element from a source element (i.e., implicit tracing). As can be observed, the algorithm splits rule matching (i.e., matching the from part of each rule with the available source elements) and rule application (i.e., evaluating the bindings of each matched rule), and most importantly trace links are initialized in the matching phase, so that every correspondence is available in the rule application phase.

The description of ATL presented above shows that a M2M language with implicit scheduling must provide some construct to describe dependencies among rules (bindings in the case of ATL), but this mechanism must avoid making the rule execution order explicit. In the following, four implicit scheduling mechanisms are briefly discussed.

- *Binding-based*. This is the mechanism used by ATL and presented in the example above. The ATL algorithm is one possible implementation, which is so simple to facilitate implementation [26]. ETL and RubyTL have the same scheduling mechanism, but in the case of RubyTL there is only one pass because rules are executed as bindings are resolved, so that traces are generated lazily.
- *Pre/post conditions*. In Relational QVT transformation rules have the form of relations between the source and target model that must hold. A relation "communicates" with other relations by specifiying *when* and *where* clauses. The when clause specifies preconditions in the form of relations that must hold for this relation to hold, whereas the where clause specifies postconditions in the form of relations that are enabled (and must hold) if this relation holds. It is important to note that pre/postconditions are different from invocations, in the sense that they do not specify that a relation has to be "executed", but just state what must hold before the transformation finishes. Besides, the QVT standard do not prescribe any particular implementation.
- *Trace constraints*. Another mechanism to describe dependencies among transformation rules is to specify that some trace links satisfying certain constraints must exist for the rule to be applicable. Rules will be in charge of creating the trace links expected by other rules, as a way to pass values among rules, but without naming the rules explicitly. It is a low-level communication mechanism that, to some extent, is similar to the strategy presented for graph-based transformations. QVT core and Tefkat are two declarative languages that use this strategy. In the case of Tefkat, a fix-point algorithm is used to resolve these dependencies, since in general it is not possible to determine a rule execution order statically.
- *Correspondence links*. It is a specialization of the previous approach, typically used in Triple Graph Grammars (TGGs), which is a graph-based technique for defining a bidirectional correspondence between source and target models. In a TGG there is a LHS and a RHS, but they are linked by correspondence nodes. These nodes ultimately act as trace links between the source elements specified in the LHS and the target elements specified in the RHS. In general to execute a TGG backtracking must be used, but under certain circunstances it is possible to avoid backtracing and resort on a fix-point algorithm [27].

2.4. *Discussion*

Given the different implicit scheduling mechanisms for M2M languages presented before, a brief comparison highlighting their pros and cons is presented in the rest of this section. To this end, four dimensions has been considered. Table I summarizes them for ATL, Tefkat, Relational QVT and TGGs.

- *Style of the language*. Transformation languages are often classified according to the programming style that they promote: imperative, declarative or hybrid. At the implementation level, it is more complicated to support flexible implicit scheduling mechanisms in non-declarative languages.

- *Scheduling mechanism*. Refers to the mechanism, at the language level, by which a developer describes the dependencies among rules.
- *Scheduling algorithm*. Refers to the algorithm by which, given a scheduling mechanism, a compiler or interpreter determine the actual rule execution order.
- *Dependency resolution*. Determining an execution order ultimately allows dependencies among rules to be resolved. This resolution is fixed when it does not take into consideration the actual shape of the dependencies (i.e., at runtime), but the mechanism (or the algorithm) forces a certain resolution scheme before rules are actually executed. On the contrary, it is called flexible when rule execution order is driven by the data-dependencies among rules.
- *Execution engine*. A model transformation can be compiled or interpreted. Some scheduling mechanisms are more suitable for compilation than others.

Dimension	ATL	Tefkat	Rel. QVT	TGG
Style	Hybrid	Declarative	Declarative	Declarative
Mechanism	Binding	Trace constraints	Pre/post conditions	Correspondence links
Algorithm	Two-pass	Fix-point	-	Backtracking/Fix-point
Dep. resolution	Fixed	Flexible	Fixed	Flexible
Execution	Compiled	Interpreted	Interpreted	Interpreted

Table I. Dimensions considered to compare scheduling mechanisms in model-to-model transformation languages.

The main issue with a two-pass iterative algorithm, such as the one of ATL, is that it is not really driven by data-dependencies among transformation rules, but the algorithm leads to a fixed dependency resolution, in the sense that regardless of the shape of the dependencies, all trace links must be computed at the beginning. Tefkat and TGGs are more flexible in this sense as dependencies are resolved dynamically depending on the values of the trace links, that may be contributed by distinct rules. This would allow, for instance, to specify transformation module interfaces as the expected data dependencies (e.g., the provided/required trace links). In the case of Relational QVT, dependency resolution is considered fixed because relations are coupled by name, so a relation cannot contribute to a precondition dynamically.

The main problem with fixed scheduling is that it hinders both interoperability with other languages that may resolve data dependencies differently and transformation composition that may require interchanging data between transformations. An additional advantage of a flexible approach is that it has the potential of dealing with streaming model transformations, in which the source model is not completely available at the beginning.

While Tefkat and TGGs appear as more flexible approaches, their engines are interpreted [†] and they rely on fix-point algorithms that tend to become inneficient as the size of the model grows. On the contrary, ATL is compiled to a special bytecode executed by the ATL Virtual Machine. It is expected that a compiled approach performs better than an interpreted one.

Regarding the style of the language, ATL is a hybrid language (although the scheduling part is mostly devoted to the declarative part). The other languages are declarative. It could be said that a flexible scheduling is constrained to declarative languages. However, in some cases imperative features are desirable, but without the cost of losing flexibility at the scheduling level.

There are therefore several trade-offs when designing and implementing an scheduling mechanism. Our aim has been to devise an scheduling mechanism that enables gathering what it is best of each approach. As will be shown, a continuation-based approach provides flexible resolution of dependencies without the need of a fix-point algorithm (at least in the common case), compilation, and also provides support for languages with imperative features. Section 7 presents some application scenarios and gives some performance figures.

[†]There are several TGG implementations available but as far as we know all of them are interpreted.

3. BACKGROUND

In the following we introduce the concept of *continuation* and its applicability to model transformations.

A *continuation* is a representation of the rest of the computation of a program, so that it can be saved at a given point of its execution and resumed later. In some way, it can be regarded as a "controlled goto". This concept is supported in some programming languages, for instance Scheme [9] or Scala [8]. A special kind of continuation is a *delimited continuation*, which can be informally seen as a continuation which is only partially captured, that is, it does not span the entire program but only a part of it. One way of describing delimited continuations is by means of shift/reset operators [28]. With *shift*, one has access to the current program execution state reified as a continuation, while *reset* allows us to establish the point where control will return after a shift is encountered and to delimit the continuation. Note that resets can be nested, and a shift will return to the nearest enclosing reset. From now on, we will use continuations to refer to delimited continuations.

Figure 3 provides a graphical explanation of the shift/reset operators to describe continuations. In the first step (a), a piece of code is delimited with a reset (this is normally done with a closure, but here it has been indicated with two ballons, "reset (begin)" and "reset (end)"). When the execution reaches the shift, the rest of the computation is the code that has not been executed from the shift to the end of the enclosing reset. At this point, step (b), the rest of the computation is captured into a continuation object (called cont), and the control flow goes back to the enclosing reset. In a programming language, if the reset construct is an expression its value is the continuation object. Afterwards the execution proceeds normally with the code written after the reset. Finally, at some point the continuation object could be resumed (step (c)), which would make the control flow jump to the beginning of the computation captured in the cont object, that is, restoring the captured execution. When this computation finishes, the control flow will go back just after the point where the continuation was resumed (cont.resume() in step (c)).

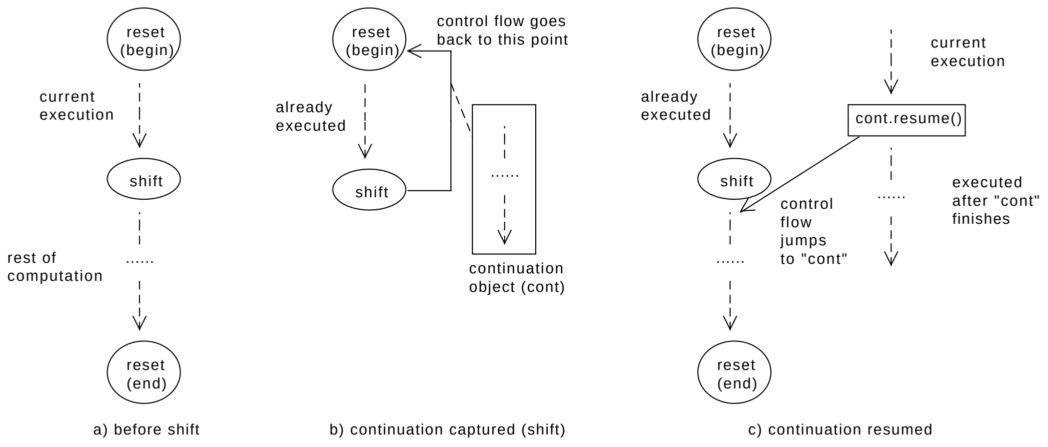


Figure 3. Graphical explanation of the shift/reset operators.

In order to illustrate continuations and its applicability to model transformations, the same transformation shown in Listing 1 is rewritten using Ruby plus continuations[‡] in Listing 3. The strategy to write this transformation using continuations has been the following.

[‡]Assuming the existence of a library with shift/reset methods. See http://chneukirchen.org/blog/archive/2005/04/shift-reset-and-streams.html for an example.

- Each function corresponds to a rule that is able of transforming one source element, passed as a parameter. A function is defined with def - end.
- Whenever a rule (i.e., a function acting as rule) needs some element produced by other rule, it just suspends its execution via a shift. The execution context (captured as a continuation) is passed to a closure as a parameter (|continuation|). This continuation is the rest of the "rule execution", and can be freely manipulated, for instance stored in some data structure for later use. The shift makes the control flow jump to the enclosing reset expression, being the continuation object captured by the shift the value of the reset expression.
- Let us also assume that there is a global associative table, called TraceModel which is filled with the relationships between a source element and one target counterpart. Its purpose will be to help resolving circular dependencies and rule communication.
- Finally, the schedule method is in charge of traversing the source model, enclosing all calls to rules into a reset to capture the continuation. Then, it performs a fixed scheduling.

```
1  def transformClass(classDcl)
2    kdm = Kdm::ClassUnit.new
3    kdm.name = classDcl.name
4
5    TraceModel[classDcl] = kdm
6
7     shift  { |continuation|  continuation }
8
9    kdmMethodUnits = classDcl.methods.map { |m| TraceModel[m] }
10   kdm.codeElements = kdmMethodUnits
11 end
12
13 def transformMethod(jMethod)
14   methodUnit = Kdm::MethodUnit.new
15   signature   = Kdm::Signature.new
16   signature.name = jMethod.name
17   methodUnit.type = signature
18   returnParameterUnit = Kdm::ParameterUnit.new
19
20   TraceModel[jMethod] = methodUnit
21
22    shift  { |continuation|  continuation }
23
24   returnParameterUnit.type = TraceModel[jMethod.returnType]
25 end
26
27 def schedule
28   pending1 = Java::ClassDcl.all_objects.map { |class_dcl|
29       reset { transformClass(class_dcl) }
30   }
31   pending2 = Java::MethodDcl.all_objects.map { |method_dcl|
32       reset { transformMethod(method_dcl) }
33   }
34   pending2.each { |continuation| continuation.call  }
35   pending1.each { |continuation| continuation.call  }
36 end
```

Listing 3: Part of the Java2Kdm expressed in Ruby with shift/reset

Next we explain the example assuming the Java model shown in 1 as source model. Both transformClass and transformMethod create a target element and use the TraceModel to associate it with the corresponding source element (lines 2-5 and 14-18 respectively). The schedule function first launches the "rule" for Java classes (lines 28-30). When class cd1 is transformed, a new KDM class unit is created, cu1, and associated through the TraceModel. The execution of the shift (line 7) makes the execution go to the enclosing reset (line 29), and the iteration continues processing cd2 similarly. The pending1 list now contains two continuations, one per rule execution. In Figure 4 these is represented as two transformClass messages whose execution is stopped, returning continuations as a result (each continuation holds a gray activation box as the rest of the computation).

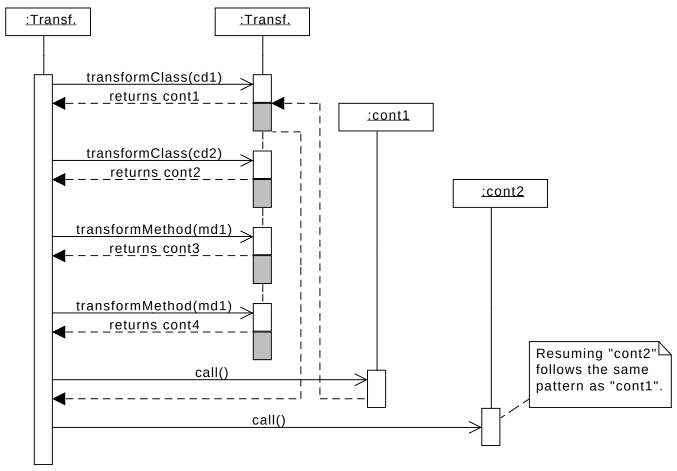


Figure 4. Example in Listing 3 partially illustrated with an adapted version of UML sequence diagrams that allows continuation-based behaviour to be represented. Activation boxes drawn in gray are the "rest of the computation" that will be captured into a continuation. Dashed arrows with full head represent control transfer messages (i.e., a goto).

The transformation of Java methods (lines 31-33) is similar (md1 and md2 are processed and transformed into {mu1,s1} and {mu2,s2}), and pending2 contains two continuations (one per transformed method). At this point, all target elements have been created and associated through TraceModel to their source counterparts, but relationships between target elements have not been set yet, in particular to resolve KDM methods from Java methods (line 9) and KDM types from Java types (line 24). Resuming the first continuation in the pending2 list means that the execution will come back to line 24, but now the TraceModel already contains the target counterpart of the method's return type. When the execution of this continuation finishes (i.e., the transformMethod finished), the control flow will return to the point where the continuation was resumed, line 34 in this case. The next continuations in the list can be resumed in the same way. Graphically, the call message over the continuation object (cont1 in the diagram) has the effect of making the control flow go to the gray activation box stored in the continuation, that is, the original transformClass message is being resumed. When this message finishes, the execution flow continues normally, in this case resuming cont2 in the same way.

This illustrates how continuations facilitates the task of waiting until a required target element is available. However, the scheduling has been hard-coded so there is little improvement with respect to, e.g. the ATL algorithm introduced before. Our aim is therefore to take advantage of the ability of a continuation to wait until a piece of data is available to devise a flexible, automatic scheduling mechanism driven by data dependencies. To this end, we propose a strategy to describe dependencies among transformation rules which is presented in the form of a simple transformation language in the next section. Afterwards, the underlying algorithm to schedule a transformation using continuations is presented.

4. LANGUAGE DESCRIPTION

This section introduces the Koan language, focusing on how dependencies among rules are described and resolved, so that implicit scheduling is seamlessly achieved. The aim of Koan has been to experiment with novel scheduling mechanisms and model transformation language features. The decision to build a new language instead of reusing an existing one is motivated by two factors. First of all, when an existing language and its engine are reused, the variety of new features and techniques that can be tried is normally limited by the original language. Secondly, we wanted to apply a bottom-up approach, building the foundations and then proceed building languages with higher-level of abstraction on top of our findings developing the low-level layers. This approach has allowed us to devise the implementation mechanism presented in this work to schedule transformation rules using continuations. As a side effect, the scheduling mechanisms built for Koan are being used as the basis to build a higher-level transformation tool [13].

Regarding the features of Koan, Figure 5 summarizes them following the terminology presented in [2] by means of a feature diagram. As can be observed, Koan has been designed as a hybrid rule-based transformation language, where the declarative part is given by transformation rules, rule dependencies specification and implicit scheduling. The imperative part corresponds to simple OO constructs: setting and getting properties, and method calls. Method calls are complemented with closures to enable navigation of multivalued relationships (e.g., using OCL-style operations like select, collect (map in the example), etc.). Rules are selected non-deterministically, but only in the sense that the execution order does not matter since a rule will be stopped if the required data is not available. Finally, it does not support incremental update (only a new target or destroying an existing target is allowed) and it is unidirectional.

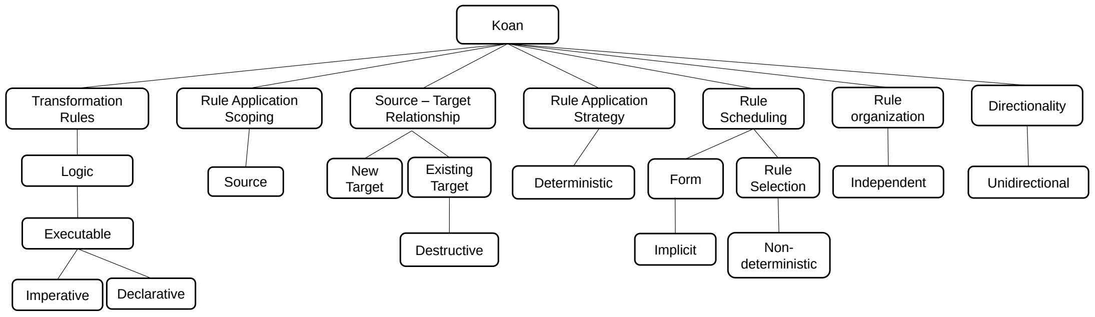


Figure 5. Koan features according to the [2].

A transformation definition is composed of rules, a trace specification, and decorators (Figure 6 shows the abstract syntax). A Decorator is a method attached to some metaclass which is extended at runtime. A TraceLink defines a relationship between one or more source elements to one or more target elements, which may contain additional information. A TraceSpecification is the set of trace link types, whose instances will conform the transformation trace model at runtime. In our case, trace links are defined as metaclasses (they inherit from the class meta-metaelement, which is EClass in the Ecore meta-modeling framework [11]), so that trace link instances are treated as regular model elements. Trace link properties are just structural features (attribute or non-containment reference). Transformation rules match elements according to some source pattern. A Rule has a Matcher that establishes which source elements must be evaluated by the rule. Because the aim of Koan has not been to innovate in pattern matching, we have relied on a simple mechanism: the ForAll matcher, which matches every instance of a given metaclass (in the style of ATL). A filter expression can be established to rule out some of the matched elements. The body of a rule is composed of a sequence of Statements (Expression also inherits from Statement but it is not shown), such as MethodCall, Closure, or Define to bind a value to a name. There are also language constructs to manipulate trace link instances as well, which are explained later.

Listing 4 is an excerpt of the preamble of the Java2Kdm transformation written in Koan. First, the set of trace links used by the transformation are declared, in this case type2datatype and method2method (lines 3 - 14). Note that a Java method is related to two KDM entities, as explained before. Then, a decorator to extend the ClassDcl metaclass with the methods facility is defined (lines 16-18). Finally, transformation rules are specified. In this case, class2classUnit transforms every Java ClassDcl into a ClassUnit, assigning the name property.

```
1  transformation java2kdm(java) −> (kdm)
2
3  trace t_java2kdm
```

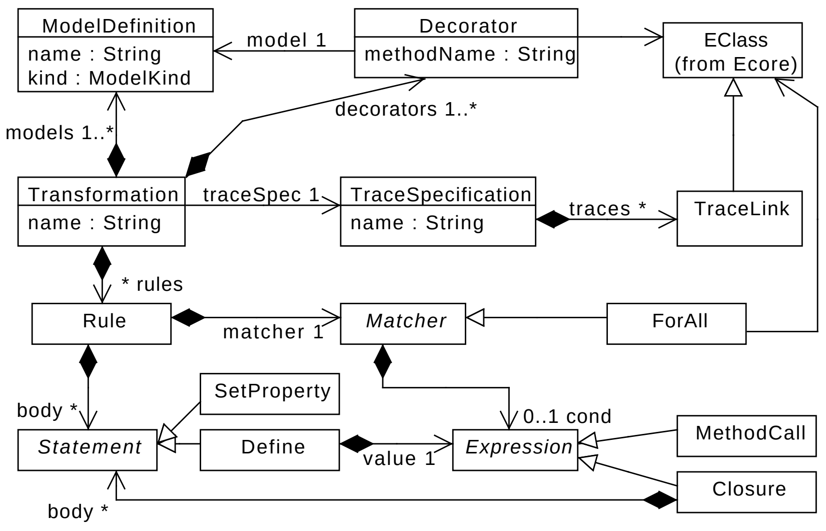


Figure 6. Excerpt of the abstract syntax metamodel for the Koan language.

```
4    link type2datatype
5      var javaType : java!TypeDcl
6      var kdmType : kdm!DataType
7    end
8
9    link method2method
10     var javaMethod : java!MethodDcl
11     var kdmMethod : kdm!MethodUnit
12     var signature  : kdm!Signature
13   end
14 end
15
16 def java!ClassDcl.methods
17   self.bodyDcls.select { |dcl| dc.kind_of(java!MethodDcl) }
18 end
19
20 rule class2classUnit
21   forAll classDcl : java!ClassDcl
22
23   classUnit = kdm!ClassUnit.new
24   classUnit.name = classDcl.name
25 end
```

Listing 4: Beginning of the Java2Kdm transformation

As can be seen, Koan features a simple OO expression language, with the characteristic that "variable assignments" bind a name to a value which cannot be rebound (line 23). It can be seen as syntactic sugar for *let-in* construct of many functional languages. Please note that that property assignments do change the object (line 24). An important feature is *closures* support (code blocks defined with { }), which along with a library of collection methods (e.g., select in line 17), enables model navigation in a way similar to OCL [29]. The rest of the section is focused on our approach to describe dependencies among transformation rules.

4.1. Describing dependencies

As explained, each M2M language must provide a mechanism to establish dependencies among transformation rules (e.g., ATL bindings, pre/post conditions in QVT Relational, etc.). In our case, we would like to describe dependencies explicitly but independently of the rules that produce the values (i.e., a rule call mechanism yields to explicit scheduling) so that rule execution is driven by these dependencies, using continuations as the underlying implementation mechanism.

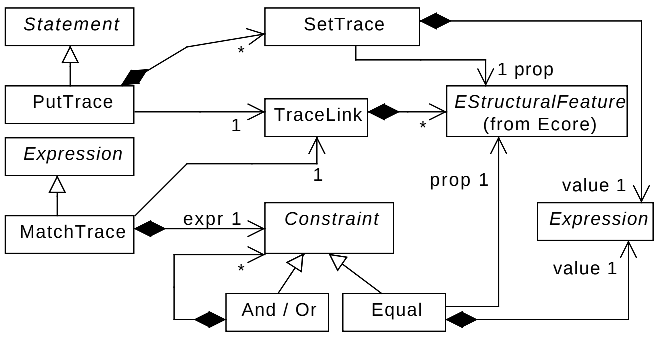


Figure 7. Abstract syntax of trace handling constructs. The reference from TraceLink to EStructuralFeature is inherited from EClass (shown for clarity).

We propose to describe dependencies among rules by setting constraints over the trace model, in the style of Tefkat [5]. In this way, dependencies are explicit, but they are decoupled from the actual rules that provide the values that satisfy them. This means that several rules may contribute the same constraint, or that a rule can be seamlessly replaced by another rule, as far as it is able of providing the same trace links, therefore, promoting modularity.

We have defined two constructs to deal with the trace model, PutTrace and MatchTrace, whose abstract syntax is shown in Figure 7. The PutTrace construct creates a new trace link, setting its properties. So, it is composed of a sequence of SetTrace instructions to establish the trace link properties. The set instructions are executed atomically, in the sense that the new link is issued to the trace model only when all of them have finished.

The MatchTrace construct is the core of our mechanism. It allows a constraint on the trace model to be specified in order to retrieve a trace link from the trace model that satisfies it. Thus, the Constraint metaclass is the root of a small declarative language to specify such constraints. In particular, Equal matches a specific feature with the value given by an expression. For the sake of simplicity, here we consider matching only one trace link, but it can be generalized to many. A MatchTrace is an expression that evaluates to a trace link that satisfies a given constraint. If there is no one, it suspends its execution until it is available.

In this way, PutTrace can be seen as a producer and MatchTrace as a consumer of trace links. Being our mechanism based on continuations, the execution state of any piece of code that depends directly or indirectly on a trace link that is required through a MatchTrace statement is stored into a continuation, that is resumed when the constraint is satisfied and the element is available. More details about this aspect are given in Section 5.

Listing 5 shows the transformation of Listing 3 written in Koan with two modifications: UnresolvedTypeDcls are also transformed to a ClassUnit tagged as "unresolved", and MethodDcls with type parameters are filtered out. Line 7 creates a type2datatype link associating the matched Java class with the newly created KDM class unit. Then, lines 10-13 are in charge of traversing the methods of the Java class, retrieving the method2method trace link associated to each one, obtaining the corresponding KDM methods (note that the chained map gets the kdmMethod property from each retrieved trace link). Next, more dependencies are resolved. It is worth noting that the normalMethod rule is in charge of associating Java methods with KDM method units (line 40-42), but also creates a circular dependency with class2classUnit via matching the type2datatype trace link (line 44-45).

```
1   rule class2classUnit
2     forAll classDcl : java!ClassDcl
3
4     classUnit = kdm!ClassUnit.new
5     classUnit.name = classDcl.name
6
7     putTrace type2datatype
8         with javaType = classDcl, kdmType = classUnit
9
10    classUnit.codeElement = classDcl.methods.map { |m|
        match method2method
           with javaMethod = m
     }.map { |trace_link|  trace_link .kdmMethod }

   # Another dependency
   superclass_tlink  = match type2datatype
                            with javaType = classDcl.superclass
   ...
end

rule unresolvedTypeDcl
   forAll unresolved : java!UnresolvedTypeDcl

   classUnit = kdm!ClassUnit.new
   classUnit.name = classDcl.name + "[unresolved]"

   putTrace type2datatype
        with javaType = classDcl, kdmType = classUnit
end

rule normalMethod
   forAll methodDcl : java!MethodDcl
    where methodDcl.typeParameters.isEmpty()

   methodUnit = kdm!MethodUnit.new
   signature   = kdm!Signature.new
   signature.name = methodDcl.name
   methodUnit.type = signature

   putTrace method2method
         with javaMethod = methodDcl,
              kdmMethod = methodUnit, signature = signature

    trace_link  = match type2datatype
                      with javaType = methodDcl.returnType
   signature.type =  trace_link .kdmType
end
```

Listing 5: Excerpt of the Java2Kdm transformation in Koan

Figure 8(a) shows how transformation rules are related through trace links plus PutTrace and MatchTrace instructions. Figure 8(b) shows an excerpt of a possible execution of this transformation using the source model depicted in Figure 1. First, the class2classUnit is applied over the cd1 : ClassDcl element. A ClassUnit, cu1, is created and associated to the cd1 source element via the type2datatype trace link. Next, the rule attempt to retrieve the counterpart of md1 (a MethodDeclaration) via a method2method trace link, but it is still not available, so it suspends (big gray activation box) until it is available. Then, the normalMethod rule is executed, which puts the trace required by the suspended execution of class2classUnit. This can be detected, resuming such rule immediately, as shown. The rule could continue, for instance, trying to retrieve the target counterpart of the source class supertype (cd2), suspending again because the value it is not available.

There is thus no need for explicitly setting an execution order, in the sense that if the source pattern is matched the corresponding rule could be applied in this precise moment or later. The data dependencies of such rule with another rules will determine in which moments the rule will be executed (it is likely that a rule execution has to be suspended and resumed several times).

The ability of suspending the execution of a rule until a trace element satisfying a constraint is available provides flexibility, but raises three concerns related to *stuckness*, that is, when a constraint cannot be resolved (i.e., no put trace provides a trace satisfying the constraint). One concern is that suspending the whole rule would prevent non-dependents parts of it from being executed if the match instruction gets stuck. To tackle this, our engine partitions a rule into pieces of dependent expressions (the process is almost straightforward since there are no variable assignments, but named definitions) so as not to suspend a piece of code unnecessarily. In the following it will be assumed that partitions are computed statically, before executing the transformation algorithm. The

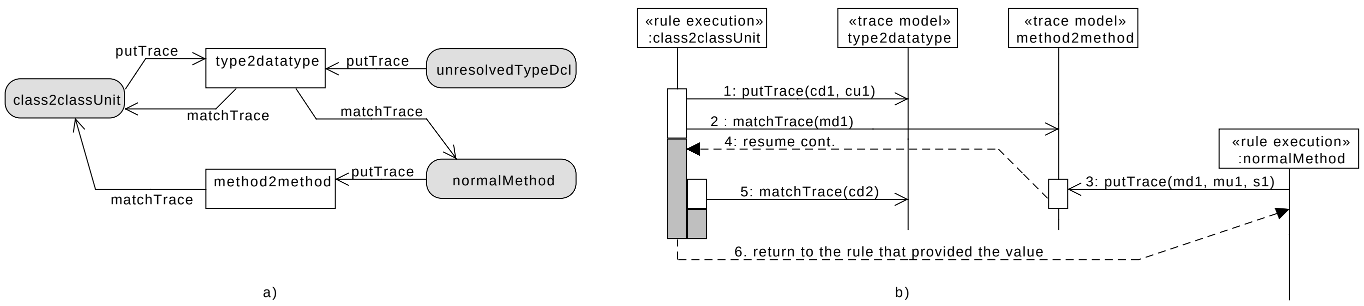


Figure 8. a) Rule dependencies through trace links. Gray rounded rectangles represents rules and white rectangles represent trace link types. b) An excerpt of an execution using the source model depicted in Figure 1. Activation boxes drawn in gray are the "rest of the computation" that will be captured into a continuation. Dashed arrows with full head represent control transfer messages (i.e., a goto)

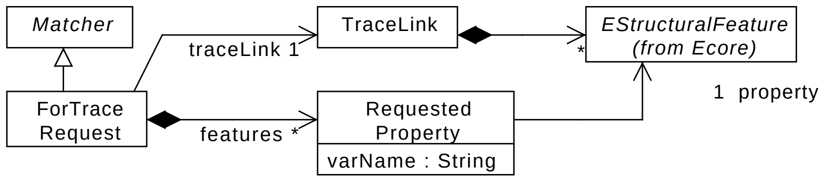


Figure 9. Abstract syntax of Koan for the ForRequestedTrace matcher

second concern is which actions should be taken when a constraint cannot be resolved. The third concern is related to using a suspendable expression (i.e., match trace) within a closure that is applied sequentially to every element of a collection (lines 10 - 13). Effectively, if a collection method operates sequentially then, if the constraint gets stuck for one of the elements of the collection, the rest of the elements get stuck as well (even though they have already been resolved). We deal with these two last concerns in Section 5.

4.2. Resolving dependencies lazily

So far we have assumed that transformation rules are always executed. However, there are cases in which one requires a transformation rule to be applied only when some of the target elements it creates are needed by another rule. This means that a rule must be triggered as a response of a dependency request. For instance, RubyTL rules [30] and QVT non-top relations [4] behave like this.

In the example, the unresolvedTypeDcl rule is always executed, but we would like not to transform unresolved Java types unless they are actually referenced by other model elements (e.g., the return type of a method may reference an unresolved type) to avoid polluting the KDM model with unneeded elements.

To tackle this issue we have defined a new type of matcher, which is able of matching a *trace request*. Figure 9 shows the corresponding abstract syntax and Listing 6 is an excerpt of the unresolvedTypeDcl rule made lazy. As can be seen a ForTraceRequest specifies the trace link type it waits for, as well as the properties expected to be constrained by the match trace constraint expression.

In a ForTraceRequest a match happens when a trace link of the declared type is requested by a match trace statement (the constraint expression must use, via Equal, all the declared trace link properties) and it is not available yet in the trace model. Such a match trace statement is suspended, and the values given in the constraint expression are put in the context of the rule using the trace link property names as variable names. Then, the "lazy rule" proceeds normally.

```
rule unresolvedTypeDcl
  forTraceRequest type2datatype with javaType = classDcl
          where classDcl.kind_of(java!UnresolvedTypeDcl)

  classUnit = kdm!ClassUnit.new
  classUnit.name = classDcl.name + "[unresolved]"

  putTrace type2datatype
     with javaType = classDcl, kdmType = classUnit
end
```

Listing 6: Defining a rule as lazy via ForTraceRequest matcher

As can be observed in the unresolvedTypeDcl rule, the type2datatype trace link is expected to have its javaType property constrained, with the filter expression ensuring that it is an UnresolvedTypeDcl element. When the rule is executed, the value used in the constraint expression will be passed as a rule parameter (classDcl in this case). The rest of the rule is the same as the original.

In general, it is responsibility of the rule to provide the requested trace with a PutTrace statement, even though it is not compulsory. For example, a lazy rule can be used as a form of "element generator" which never returns, but creates some element(s) upon request, placing them in some location of the target model. Of course, the corresponding match trace will get stuck (see Section 5.1).

5. EXECUTION ALGORITHM

In the previous section we have shown how dependencies are represented. In the best case, there exists an execution order for rules in which requested elements are always available in the trace model, so rules can be executed sequentially. However, in general, any moderately complex model transformation has dependencies that prevent such a simple execution mechanism.

The basic idea of our algorithm can be summarized as follows. For each rule, the elements that must feed it are selected according to its matcher (only ForAll matcher), executing the rule once for each element. If a particular rule execution encounters a MatchTrace expression whose constraint cannot be resolved immediately, its execution is suspended by capturing a continuation which will be resumed when the required data is available. Whenever a PutTrace statement is evaluated, it looks for continuations that are waiting for the trace link being put in the trace model, and resumes them.

Listing 7 describes an evaluator, specified using Ruby syntax, which gives the execution semantics of Koan in terms of shift/reset operations. The following points explain the algorithm.

- **Data structures**. There are two data structures shared by all rules of the transformation. a) the trace model that consists of trace link instances, as declared by the transformation definition, b) a table that associates constraints that have not been satisfied, and are currently suspended, with a continuation (it is called *CTable*).
- **Launching rules at the top level** (function top_level). For each rule with a ForAll matcher, all instances of the given metaclass are traversed. The all_forall_rules variable contains those rules, and all_objects_of returns all instances of a given metaclass.
- **Executing a rule** means putting the matched source element into an execution context, and evaluating all the statements and expressions. Each rule is split into a list of partially ordered partitions. Any statement or expression that depends directly or indirectly from a trace link obtained from a match trace expression form parts of the partition (including other match trace expressions). As explained in the previous section, this is done statically, and partitions are available through rule.partitions.
- **Evaluating a partition**. Each partition is independently evaluated because the call to its evaluation (eval_partition) is done within a *reset*. Thus, any shift will return to this point, allowing the rest of partitions to be executed.
- **Evaluating a MatchTrace** (eval_match_trace). When a match trace expression is found, the trace model is queried to check if the constraint is satisfied (call to check_trace).
  - If it is satisfied, the execution goes on normally, returning the required trace link.

  - If not, a continuation is captured (using shift) and it is kept in the CTable, associating the trace link type, the trace properties and the values, with such a continuation. Then, the control flow returns to the reset point. The continuation will be resumed only when the required trace is available, so it will be "safe" to execute line 24 (see next point).

- **Evaluating a PutTrace** (eval_put_trace). A new trace link is created according to the put trace expression (issue_trace_link creates and puts it into the trace model). Then, the CTable is accessed to look for, and remove, those continuations that are precisely waiting for the new trace link (line 29). Each suspended continuation is resumed (calling call), but enclosed in a reset in case it is suspended again because of another MatchTrace.
- When the execution finishes, dependencies that have not been resolved are stuck. The following two subsections discusses what to do with them.

```
1  def top_level
2    all_forall_rules .each { |rule|
3      all_objects_of (rule.matcher.metaclass).each { |src_element|
4        context = create_rule_execution_context(rule, src_element)
5        rule. partitions .each { | partition |
6          reset {
7            eval_partition (context, partition , src_element)
8          }
9        }
10     }
11   }
12 end
13
14 def eval_match_trace(expression)
15   if ! check_trace(expression)
16     shift { |cont|
17       CTable.put(expression, cont)
18       nil
19     }
20   end
21   # This is the continuation: What has not been done yet
22   # (between shift and enclosing reset).
23   # The return value is always the expected trace
24   return get_trace(expression)
25 end
26
27 def eval_put_trace (expression)
28   trace_link    = issue_trace_link (expression)
29   continuations = CTable.retrieve_suspended(trace_link)
30   continuations.each { |c|
31     reset {
32       c. call
33     }
34   }
35 end
```

Listing 7: Execution algorithm of Koan using a Ruby-like syntax

The algorithm does not deal with the ForTraceRequest matcher, but its inclusion is relatively simple, as shown in Listing 8. The evaluation procedure for MatchTrace, eval_match_trace, would be modified to notify that the evaluation is being performed in order to execute lazy rules if necessary. The notify_match_trace function just tries to find a lazy rule that is able of handling the MatchTrace. If found, the rule is evaluated creating a context for its execution and evaluating its partitions independently, as before.

```
def eval_match_trace(expression)
  notify_match_trace(expression)
  # ... same as before ...
end

def notify_match_trace(expression)
```

```
  rule = find_lazy_rule_for (expression)
  if rule
    src_element = extract_src (expression)
    context     = create_lazy_rule_execution_context (rule, src_element)

    # Execute the rule normally
    rule.partitions.each { | partition |
      reset {
        eval_partition (context, partition, src_element)
      }
    }
  end
end
```

Listing 8: Executing lazy rules

As can be seen, with this continuation-based approach there is no need to perform multiple passes in order to resolve dependencies, as is the case with other approaches. Besides, the algorithm does not assume any particular layout for the data dependencies, but the use of continuations enables resolving each dependency as the data becomes available. This makes the scheduling mechanism be "flexible", according to the dimensions discussed in 2.4. Besides, it works both for declarative and imperative languages, and it allows the transformation language to be compiled (as we have done in our implementation).

However, a concern with the algorithm is that under the following two situations it may yield to unexpected results: a) two or more rules obtain through a match trace the same target element, and they set the same property, and b) when two PutTrace statements put two different trace links with the same "source" properties (properties referring to source elements). Any transformation that provokes these situations is not considered valid. The transformation engine can dealt with these issues easily using a conservative approach and checking the conditions at runtime. In the case of (a) the second time a feature of a trace link is assigned an error is raised, whereas for (b) each time a new trace link is put into the trace model, the condition is checked raising an error if needed. However, this can be an expensive operation, so a possibility is to check it only in debug mode.

5.1. *Dealing with stuckness*

As stated above, an important concern is which actions should be taken when a transformation gets stuck. We say that a transformation is stuck, when at least one MatchTrace is stuck, that is, when after finishing the transformation algorithm its constraint has not been satisfied. Please note that stuckness does not necessarily mean that a transformation is wrong because it does not handle all dependencies. On the contrary, the developer could have purposely ignored the resolution of certain elements. For instance, in Listing 5 the normalMethod rule (lines 31–47) matches only those methods that are not generic (i.e., do not have a type parameter). Thus, if the source model contains a generic method it will not be transformed, so any attempt to resolve a correspondence from this method will provoke stuckness (e.g., in lines 11-12 in Listing 5). The transformation engine could notify about this situation, allowing the transformation developer to create a rule that resolves the element (e.g., dealing with methods with generic parameters) or just to ignore the situation (e.g., if generic methods are not to be transformed).

In order to deal with this issue in a more flexible way, we extended Koan to allow associating a closure to a MatchTrace expression. The closure will be called at the end of the transformation execution if the expression is stuck. The code of the closure is not allowed to execute any trace-related action (e.g., another match trace), but only to perform "clean up" actions, such as setting some default value. As a facility to reuse code, instead of a closure the name of a function could be attached. For example, Listing 9 is a modified version of the class2classUnit rule, that takes into account the possibility of not having a rule for transforming a method (e.g., methods with type parameters are not handled in this example). To this end, the on-stuck closure (lines 10–17) attaches an annotation to the created class as way to indicate to the user of the target model that some methods have not been considered. The annotation contains a reference (ref) to the element that was not resolved. In addition, the ctx parameter allows the transformation context to be accessed, in this case to log a message.

```
rule class2classUnit
  forAll classDcl : java!ClassDcl

  classUnit = kdm!ClassUnit.new
   ...

  classUnit.codeElement = classDcl.methods.map { |m|
     match method2method
       with javaMethod = m
       on−stuck { |ctx|
          unresolved = kdm!Annotation.new
          unresolved.body = m.name + "has been ignored"
          unresolved.ref  = m
          classUnit.annotations = unresolved

          ctx.log("Unresolved element!")
       }
  }.map { |trace_link| trace_link.kdmMethod }

   ...
end
```

Listing 9: Associating a clean-up closure in case of stuckness

This feature can be used to emulate the behaviour of some transformation languages, for example the implicit tracing mechanism of ATL [31]. The example above can be seen as a customized version of implicit tracing. In our case, we have taken advantage of it to output a model reporting which elements have not been resolved, so as to facilitate debugging the transformation definition if needed.

5.2. *Dealing with collection iterator operations*

The algorithm we have presented is mainly intended to resolve constraints one by one, but a transformation normally has to handle multivalued relationships by mapping elements in a source relationship to target elements in a target relationship possibly with the requirement of preserving the order. However, the algorithm presented so far only allows iterator operations to have an all or nothing semantics.

In the running example, Java methods are mapped to KDM Method units (Listing 5, lines 10-13) by traversing MethodDcl elements contained in the methods property, and setting a constraint on the method2method trace. However, the normalMethod rule is filtered, so elements not satisfying the filter expression (typeParameters.isEmpty()) will make such constraint to be stuck. If the methods property contains elements than can be resolved and also elements that will get stuck, the whole iterator expression gets stuck, because collection operations like map, select, etc., work sequentially. Thus, if one gets stuck, the next iteration never happens, and the operation never returns.

A possible workaround is to filter out those elements that are not going to be transformed according to rule filters, but this implies changing those filter expressions each time a new rule is added, hindering maintainability.

Therefore, to deal with this a special form of map has to be defined. The new version is called *cMap* and it behaves as a regular *map* except that it discards those elements whose closure has got stuck. Interestingly, the partial order is still preserved (as long as the collection has a traversal order, e.g., a list).

Listing 10 shows a possible implementation of cMap in terms of regular collection iterators and shift/reset. It takes a receptor (the collection) and a closure that will be executed for each element. The original execution algorithm has to be modified adding a new data structure, called PreStuckList which holds a list of continuations that must be executed before finishing the transformations (and considering it as stuck).

```
def cMap(receptor, closure)
```

```
2    result   = []
3    resolved = []
4    receptor.each_with_index { |obj, idx|
5      reset {
6        r = eval_closure(closure, obj)
7        resolved << idx # add to list
8        result[idx] = r   # set value at idx
9      }
10   }
11   if resolved.size != original.size
12     shift { |continuation|
13       PreStuckList.add(continuation)
14     }
15   end
16   return resolved.map { |idx| result[idx] }
```

Listing 10: Implementation of cMap with shift/reset

Given a call like classDcl.methods.cMap { match t_java2kdm::method2method with source = m } the cMap method receives the collection to be traversed (receptor) and the closure to be called for each element. The result variable keeps an initially empty list with the intermediate result, and resolved is a list of indexes that have already been resolved (to preserve order). Then, the collection is traversed one by one, so that a reset point is set in each iteration to make them independent (lines 4-10). If a particular closure execution gets stuck, it will never return its result, so lines 7-8 will not be executed, but the rest of iterations will. Finally, if no closure has been suspended, the method can return normally (i.e., when there are less results in resolved than the size of the original collection). If not, it has to wait, which is done in lines 12-14, adding the continuation to the PreStuckList list. Before actually finishing the transformation, the continuations in the PreStuckList list are resumed effectively discarding those elements that have not been resolved (line 16). Please note that the PreStuckList has to be iterated until it is empty (in a fix-point fashion), because a new continuation could be added to the list after having resumed a cMap. This would happen for instance if there are two cMap operations where the output of the first is the input of the second, and the first one gets stuck.

Other iterators such as *select* can be built in a similar way. For other operations such as *iterate* (*foldLeft* in some functional languages) it is not possible to apply this solution, since each iteration depends on the previous one. This could be regarded as a limitation, but actually the natural behaviour is to stuck the whole iterate expression. The rationale is that, if the value that must be passed to the next iteration depends on the result of a MatchTrace that gets stuck, the whole expression must be naturally get stuck because there is an implicit computation order. Finally, a useful alternative version for cMap is uMap, that does not preserve the order, so it does not require the pre/post-processing with the resolved intermediate list.

6. RELATED WORK

Many transformation languages have been developed over the last years and some of their algorithms are documented in the literature. Graph transformation algorithms are typically focused on pattern matching (for example, VIATRA2 [16] adapts the RETE algorithm, and GrGen.NET [32] has adaptive search plans), leaving scheduling to some dedicated language. An exception is AGG [17], which features implicit scheduling by rewriting the graph until a fix-point is found. AGG uses critical pair analysis to detect conflicts between rules that could lead to a non-deterministic result. In general, critical pair analysis has been used to detect conflicts in rule systems, so we believe that it could be adapted for our case, to detect conflicts among PutTrace statements and to prevent indeterminism.

Regarding transformation languages specialized in model-to-model transformations (in contrast to in-place transformations) we review in detail three relevant types of languages with implicit scheduling, and discuss how our approach can be used to implement each one, thus showing the usefulness of our work beyond Koan. The explanation is given as a mapping to Koan constructs, but each particular implementation will need to adapt our algorithm to deal with the specific constructs of each language.

ATL. It is a hybrid model-to-model transformation language based on the concept of matched rule, that consists of a source pattern, a target pattern, and a set of *bindings* relating target properties to source elements. Bindings are implicitly resolved by rules. The original ATL algorithm is documented in [31]. It has two fixed phases: the first one matches the source model with the source patterns of the rules, creating the corresponding target elements and traceability links (they form an implicit trace model). The second phase traverses the traceability links, initializing the target properties of elements according to the bindings (this implies looking up correspondences in the trace model).

It is possible to implement a similar behaviour with our approach. The trace model would consist of only a generic type of trace link, with as many elements (of type Object) as declared in the matched rule with the most source and target elements. ATL matched rules correspond to our notion of "forall" rule. The first part of the rule execution would create the target elements (without initializing their features), and a PutTrace would relate the matched source elements with the target elements. Then, every binding of the rule would correspond to a MatchTrace. An important issue is the *implicit tracing* feature of ATL, which means that ATL only resolves a trace link if it is actually transformed. If not transformed, implicit tracing just returns the source element on which the tracing was attempted (setting the target property with the source element if the type is compatible). This case can be handled using a mechanism similar to the one presented in Section 5.1 to deal with stuckness. Another feature that could be supported is *unique lazy rules*, which can be roughly emulated with our notion of "lazy rule".

QVT. The QVT standard does not impose any algorithm neither for Relational or Operational QVT, but tool vendors are free to choose. The Relational QVT language is a declarative bidirectional language based on establishing relations between models and featuring change preservation. Here we consider execution in one direction, that has to be complemented with a mechanism to look up a persistent trace model in order to preserve changes and provide a proper synchronization. Top relations would be mapped to "forall rules" and normal relations to "lazy rules". For each relation, there would be a trace link type relating the objects matched by the rule domains. In QVT scheduling must be done with respect to the pre/postconditions of the relations. Preconditions would yield to a sequence of MatchTrace statements (suspending the rule execution until the needed data is produced). Then, there would be some logic to create/update target elements from the source domain(s) and to link them through PutTrace statements. Executing a relation in a postcondition would simply map to another MatchTrace statement, that would yield to the execution of a "lazy rule" (relation) or to wait for a "forall rule" (top relation) to put the corresponding trace.

On the other hand, QVT operational uses external rule scheduling but has the *late resolve* feature to defer the resolution of a correspondence until the end of the transformation. This could seamlessly be implemented with continuations.

Tefkat. It is a logic-based declarative transformation language [5]. Our approach to describe rule dependencies is partly inspired by Tefkat's, where the trace model is called *tracking extent*, and rules allow LINKS and NOT-LINKS clauses to describe preconditions setting constraints on the tracking extent. LINKING clauses allow a tracking class to be instantiated setting its properties. Tefkat computes an execution order statically via stratification, however rules within the same stratum require re-execution until a fix-point is found. This may be inefficient because of the need of rebuilding the evaluation tree.

Adapting our approach to Tefkat is straightforward in the case of LINKS and LINKING clauses that correspond to MatchTrace and PutTrace respectively. Our algorithm could be adapted to take into account the execution order that Tefkat computes statically. The main issue is that we do not support negative trace matches (NOT-LINKS in Tefkat), but it could be supported by a similar mechanism to the cMap implementation under the restriction that a continuation resumed from the PreStuckList must not create trace links of the type specified by the NOT-LINKS clause (which in most cases will be plausible).

Other languages with implicit scheduling are ETL [23] and RubyTL [30]. They share several commonalities to ATL, so the adaptation of our approach to them is alike. In Section 2 approaches for in-place model transformation have already been presented. In principle, our approach is not applicable to the scheduling part of those languages with explicit scheduling, although we are looking into the possibility of embedding our approach on a graph rewriting engine. This endeavour also includes applying continuations to TGGs.

On the other hand, live transformations are related to the idea of transformations that wait for data to be available. A live transformation is a type of incremental model transformation where the transformation context is kept active so that the target model is updated whenever a change happens in the source model. A live transformation however is never suspended, but it typically contains event handlers that listen for changes. It could be possible to used some of the techniques presented here to implement certain live transformations schemes. In the context of graph transformations a trigger-action approach is proposed for VIATRA2 [33], where the execution context basically consists of maintaining the pattern matcher state. In this case, there is no implicit scheduling but it is explicit as with regular VIATRA2 transformations, so our approach does not apply. Please note that triggers are basically event handlers which execute a rule from scratch, while a continuation retains the state of a rule that has started its execution. In [34] live transformations are studied for Tefkat. The transformation context is the resolution tree of the Tefkat's logic-based engine. Changes in the input models are propagated to changes in the tree which in turn provoke changes in the target models. Keeping the resolution tree is costly, since it records all dependencies. In [35] an approach to live transformations for the declarative part of ATL is presented. The ATL engine is modified to record an execution trace, and the original algorithm is modified to create event listeners for each rule and binding. In the case of Tefkat and ATL, continuations could be used to suspend a rule that depends on another rule until the proper change happens. Besides, current approaches to live transformations are for languages with explicit scheduling (e.g., VIATRA2) or for purely declarative languages. Continuations could provide a means to implement live transformations for other style of languages, e.g., imperative transformation languages.

Finally, with respect to the notion of continuation, it has been applied to various domains such as compiler construction [6], web servers [7], concurrency [8], etc. However, as far as we know, this is the first work addressing its applicability to model transformations

7. EVALUATION

Our proposal is validated in several ways. First of all, we have implemented a full compiler from Koan to the JVM, which demonstrates the feasibility of our proposal. We have used the JavaFlow library which provides support for continuations at the JVM level. The implementation has been tested by developing several examples of model transformations. In particular, we have re-implemented the Java2Kdm reverse engineering transformation which is a complex ATL transformation included in the MoDisco tool [12]. This shows that it is possible to tackle real cases with a continuation-based scheduling mechanism. Secondly, we have identified and we are experimenting with some applicability scenarios in which our approach is proving useful. Thirdly, we have performed several benchmarks to assess the performance of our algorithm along with the Koan prototype implementation.

7.1. Applicability

We have identified four scenarios in which our continuation-based approach could be useful. They are briefly presented below.

7.1.1. Transformation modularity A modularity mechanism for a M2M language require means to decompose a transformation into modules and then to compose those transformation modules. When composing two transformations it is likely that one depends on the data generated by the other (sometimes even circularly). Thus, both transformations has to be scheduled together, but this is not always possible if the scheduling algorithm is fixed.

A continuation-based approach would allow a transformation to gather elements generated by the other transformation, since it just would “wait” for such elements to be generated. In our approach, both transformations would need to agree on the trace model, which would act as the interface between transformations.

7.1.2. Transformation language interoperability Transformation language interoperability has been regarded as an important topic in model transformation [36]. However, so far, only limited forms of interoperability has been achieved [37].

Our proposal for transformation language interoperability is to compile each language to a common runtime (a variant of Koan in this case), based on continuations. Two languages with different scheduling mechanisms (e.g., explicit and implicit), could interchange data just generating MatchTrace and PutTrace instructions to request the values that one transformation is interested in, and to provide such values respectively. Our initial results in this respect are reported here [13]. We have been able to build four languages of different nature (mappings, target-oriented transformations, attribute computation and simple pattern matching) and use them to split a transformation into several parts, each one addressed with one language.

7.1.3. Streaming model transformations An streaming model transformation is a special kind of transformation in which the source model is not available at the very beginning, but it is streamed as it is produced. The transformation engine is requested to generated the target model as the new source elements appears.

As far as we know, currently no M2M language supports streaming model transformations, as there are several issues that must be addressed, such as the life span of the source elements and the trace model or which expressions are no longer valid (e.g., allInstances in OCL). An important issue is that the scheduling algorithm must take into account that the resolution of a given element must wait until the corresponding value is streamed. A two-pass algorithm is thus not applicable.

We have started applying our approach to this kind of transformations, in particular trying to addresss the resolution of elements that have not already been streamed. In [38] we show a simple example, which is naturally solved with our approach, since a transformation rule just suspends itself into a continuation, and it is resumed when the required element appears in the stream.

7.1.4. Execution platform Regarding the usability of Koan, it can be said that in general is too low level, in the sense that some care has to be taken in certain aspects. For instance, sometimes forgetting to write a PutTrace relating the source and target elements of a rule, will provoke another MatchTrace to get stuck, even though the rule for it has been written. Another example is the need to resolve multivalued references “manually” using cMap.

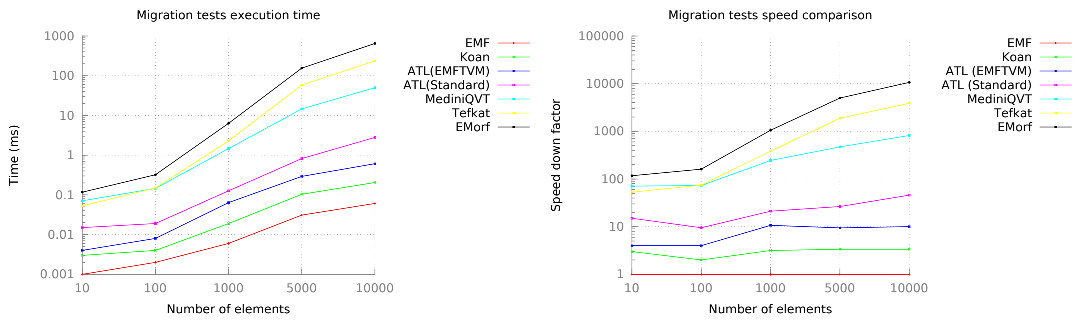
Hence, we consider Koan as a proof-of-concept that can be used as a low-level language to which a high-level language can be compiled to. This would relieve the transformation language developer from dealing with continuations, but he or she would just compile high-level transformation constructs to primitive Koan instructions.

7.2. *Performance evaluation*

Our continuation-based algorithm enables advanced transformation scenarios to be addressed due to its flexibility. To assess whether this flexiblity comes at the cost of peformance penalities we have run peformance benchmarks to compare against other languages. We have taken the benchmark proposed in [39] to compare model migration tools. It consists of transforming a PetriNet model to another model that conforms to a variation of the original metamodel. The rationale to choose this benchmark is that this model migration in particular implies little model navigation, but the structure of the PetriNet model requires resolving several (sometimes circular) dependencies. We have implemented this transformation in EMorf (a TGG engine), Tefkat, MediniQVT (Relational QVT), ATL, Koan, and Java/EMF (the scheduling has been hard-coded, trying to avoid traversing

Num. elems.	EMorf	Tefkat	MediniQVT	ATL (Standard)	ATL (EMFTVM)	Koan	EMF
10	0.117	0.053	0.071	0.015	0.004	0.003	0.001
100	0.322	0.149	0.146	0.019	0.008	0.004	0.002
1000	6.330	2.306	1.471	0.127	0.064	0.019	0.006
5000	154.983	58.373	14.641	0.820	0.292	0.104	0.031
10000	648.378	237.407	54.080	2.797	0.611	0.205	0.061

Table II. Execution time (in seconds) for the migration benchmark



(a) Comparison of execution times (b) Comparison of “speed-down” (w.r.t. EMF)

Figure 10. Results of the performance benchmarks as comparative charts.

the same elements whenever possible, in order to achieve the best performance)[§]. Two versions of the ATL Virtual Machine(VM) has been considered, the standard ATL VM and a new version called EMFTVM [37]. The reason is that, as will be shown, the standard VM is more inneficient than the new version.

The results of the execution for the input models provided by the original benchmark are shown in Table II, and graphically summarized in Figure10(a). Results do not include neither loading nor serializing time. As can be seen, Koan is faster that the other languages analyzed and scales better. Moreover, compared the performance results obtained in [39] for several migration tools, Koan is faster than those specialized engines. A possible bias with Tefkat and MediniQVT is that their matching mechanisms are more complex that Koan and ATL, so some of their execution time is probably spent there. Regarding to the difference with ATL, in principle the comparison of the algorithms suggests that Koan should outperform ATL by a factor of 2. In fact, the empirical result for ATL EMTVM is close to this, and probably the difference lies on low-level implementation issues (e.g., we compile Koan to the Java VM directly, while EMFTVM is built on top of the Java VM).

The results for Java/EMF show the minimum execution time that an engine could achieve, so that the time difference between Koan and Java/EMF can be considered mostly due to the Koan scheduling algorithm overhead (note that both of them compile to JVM bytecode). Figure 10(b) shows the loss of performance with respect to the EMF implementation (we have called this “speed-down”), so that the smaller loss the better. This result means that the scheduling algorithm is a time consuming part (around 60%). Therefore, optimizations to detect those cases where resolving a MatchTrace can be done with just a plain method call would speed up Koan even more. In any case, the overhead is “fixed” in the sense that Koan scales quite well as the number of elements increases, in fact, the scale factor is similar to Java/EMF and ATL EMFTVM version.

In order to assess the performance of Koan with a more complex case study, we have compared our implementation of the Java2Kdm transformation against the original ATL version (executed with the standard VM, since the EMFTVM version is not able yet to deal with such transformation). Although do not claim that our version is exactly equal to the ATL version, the benchmark gives

[§]The benchmarks have been run in an Intel Core2 Duo E6550, with 2GB RAM, under Eclipse 3.6 configured with a heap size of 1GB. We have followed the recommendations for benchmarking Java programs given in [40].

	Size (MB)	Num. elems.	ATL (Standard)	Koan
JAX-RS	2.2	11270	12.516	0.769
Javaflow	6.2	22412	24.597	1.958
Robocode	7.2	41107	33.683	2.824
UMLet	26.0	108246	822.312	8.124

Table III. Execution time (in seconds) for the Java2Kdm benchmark applied to four open source projects, comparing ATL (standard VM) and Koan.

us a comparative hint about the scalability of our approach. Four open source projects have been used as input models: (1) Javaflow (the continuation library we have used), (2) Robocode (an API to develop tanks that fight in virtual battles), (3) UMLet (a UML tool), and (4) JAX-RS (a framework to develop REST services). Using MoDisco an abstract syntax model of each project is obtained, the size of the model stored as an XMI file and the number of elements indicates how large is each model. Then, both ATL and Koan transformations have been benchmarked for each model.

As can be seen, Koan is significantly faster in all cases. This benchmark demonstrates that our approach is also able of handling a real, industrial case in an efficient way.

8. CONCLUSIONS

In this paper a novel implicit scheduling mechanism based on continuations has been reported. An execution algorithm based on shift/reset operations has been given, and additional issues have been tackled, namely lazy rule execution, stuckness, and iterator operations. This algorithm is the core of the Koan transformation language which has also been presented. Altough we have used Koan as a means to experiment with continuations and scheduling, our proposal is not specific to it, but it is intended to be part of the implementation of other transformation engines. In this sense, insights about how it can be used to implement other languages have been given.

Our work has been validated in several ways. First of all, we have implemented a compiler from Koan to the JVM, which proves the feasibility of our approach. It has been tested by developing some model transformations. We are also experimenting with advanced features, such as transformation composition, language interoperability and streaming transformations. Our experiences suggest that the flexibility provided by our approach is adequate to tackle these features. Finally, performance benchmarks show that our approach is efficient and scalable, and even outperforms state-of-the-art transformation languages. The results reported on this work are the basis for our project to build a model transformation tool, which is freely available at `http://sanchezcuadrado.es/projects/eclectic`.

As future work we plan to continue applying continuations to advanced features, particularly to streaming model transformations. We also plan to improve our implementation, and also building more test transformations.

ACKNOWLEDGEMENT

This work was funded by the Spanish Ministry of Economy and Competitivity (project “Go Lite” TIN2011-24139) and the R&D programme of the Madrid Region (project “e-Madrid” S2009/TIC-1650).

REFERENCES

1. Mens T, Gorp PV. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science* March 2006; **152**:125–142, doi:10.1016/j.entcs.2005.10.021.
2. Czarnecki K, Helsen S. Feature-based survey of model transformation approaches. *IBM Syst. J.* July 2006; **45**:621–645.
3. Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. *Science of Computer Programming* 2008; **72**(1-2):31 – 39.
4. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation 2005.
5. Lawley M, Steel J. Practical Declarative Model Transformation with Tefkat. *Model Transformations in Practice Workshop of MODELS'05* 2005; .

6. Appel AW. *Compiling with Continuations*. Cambridge University Press, 1992.
7. Ducasse S, Lienhard A, Renggli L. Seaside: A flexible environment for building dynamic web applications. *IEEE Software* 2007; **24**:56–63, doi:http://doi.ieeecomputersociety.org/10.1109/MS.2007.144.
8. Rompf T, Maier I, Odersky M. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. *Proceedings of the 14th International Conference on Functional Programming*, 2009; 317–328.
9. Adams NI IV, *et al.*. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.* September 1998; **33**:26–76.
10. Thomas D, Hunt A. *Programming Ruby: The pragmatic programmer's guide*. Addison-Wesley, 2000.
11. Steinberg D, Budinsky F, Paternostro M, Merks E. *EMF: Eclipse Modeling Framework 2.0*. 2nd edn., Addison-Wesley Professional, 2009.
12. Bruneliere H, Cabot J, Jouault F, Madiot F. Modisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. *ASE*, 2010; 173–174.
13. Cuadrado JS. Towards a family of model transformation languages. *ICMT*, *LNCS*, vol. 7307, Springer, 2012; 176–191.
14. Kleppe A. Mcc: A model transformation environment. *Model Driven Architecture - Foundations and Applications: Second European Conference, ECMDA-FA'06*, *LNCS*, vol. 4066, Springer Verlag, 2006; 173–187.
15. OMG. Architecture-driven modernization (adm): Knowledge discovery meta-model (kdm), v1.1 2009.
16. Varró D, Balogh A. The model transformation language of the viatra2 framework. *Sci. Comput. Program.* October 2007; **68**:187–207.
17. Taentzer G. AGG: AGraph Transformation Environment for Modeling and Validation of Software. *Applications of Graph Transformations with Industrial Relevance*, *LNCS*, vol. 3062. Springer, 2004; 446–453.
18. Arendt T, Biermann E, Jurack S, Krause C, Taentzer G. Henshin: Advanced concepts and tools for in-place emf model transformations. *Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, vol. 6394, Petriu D, Rouquette N, Haugen (eds.), Springer Berlin Heidelberg, 2010; 121–135, doi: 10.1007/978-3-642-16145-2_9. URL http://dx.doi.org/10.1007/978-3-642-16145-2_9.
19. Lara J, Vangheluwe H. Atom3: A tool for multi-formalism and meta-modelling. *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, vol. 2306, Kutsche RD, Weber H (eds.), Springer Berlin Heidelberg, 2002; 174–188, doi:10.1007/3-540-45923-5_12. URL http://dx.doi.org/10.1007/3-540-45923-5_12.
20. Schrr A. Specification of graph translators with triple graph grammars. *Graph-Theoretic Concepts in Computer Science*, *Lecture Notes in Computer Science*, vol. 903, Mayr E, Schmidt G, Tinhofer G (eds.), Springer Berlin Heidelberg, 1995; 151–163, doi:10.1007/3-540-59071-4_45. URL http://dx.doi.org/10.1007/3-540-59071-4_45.
21. Börger E, Stärk R. *Abstract state machines: a method for high-level system design and analysis*. Springer, 2003.
22. Fischer T, Niere J, Torunski L, Zndorf A. Story diagrams: A new graph rewrite language based on the unified modeling language and java. *Theory and Application of Graph Transformations*, *Lecture Notes in Computer Science*, vol. 1764, Ehrig H, Engels G, Kreowski HJ, Rozenberg G (eds.), Springer Berlin Heidelberg, 2000; 296–309, doi:10.1007/978-3-540-46464-8_21. URL http://dx.doi.org/10.1007/978-3-540-46464-8_21.
23. Kolovos D, Paige R, Polack F. The epsilon transformation language. *Theory and Practice of Model Transformations*, *LNCS*, vol. 5063, Springer, 2008; 46–60.
24. Cuadrado JS, García J, Menárguez M. RubyTL: A practical, extensible transformation language. *Model Driven Architecture - Foundations and Applications: Second European Conference, ECMDA-FA'06*, *LNCS*, vol. 4066, Springer, 2006; 158–172.
25. Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. *Science of Computer Programming* 2008; **72**(1-2):31 – 39.
26. Jouault F, Kurtev I. Transforming models with atl. *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005. URL http://sosym.dcs.kcl.ac.uk/events/mtip/submissions/jouault_kurtev__transforming_models_with_atl.pdf.
27. Giese H, Hildebrandt S, Lambers L. Bridging the gap between formal semantics and implementation of triple graph grammars. *Software & Systems Modeling* 2012; :1–27.
28. Danvy O, Filinski A. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science* 1992; **2**(4):361–391.
29. Warmer JB, Kleppe AG. *The Object Constraint Language: Precise Modeling With UML*. Addison Wesley, 1998.
30. Sanchez Cuadrado J, Garcia Molina J. Modularization of model transformations through a phasing mechanism. *Software and Systems Modeling* 2009; **8**:325–345.
31. Jouault F, Kurtev I. Transforming models with atl. *Satellite Events at the MoDELS 2005 Conference*, *LNCS*, vol. 3844, 2006; 128–138.
32. Gei R, Kroll M. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. *Applications of Graph Transformations with Industrial Relevance*, *LNCS*, vol. 5088. Springer, 2008; 568–569.
33. Rth I, Bergmann G, krs A, Varr D. Live model transformations driven by incremental pattern matching. *Theory and Practice of Model Transformations*, *Lecture Notes in Computer Science*, vol. 5063, Vallecillo A, Gray J, Pierantonio A (eds.), Springer Berlin / Heidelberg, 2008; 107–121.
34. Hearnden D, Lawley M, Raymond K. Incremental model transformation for the evolution of model-driven systems. *Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, vol. 4199, Nierstrasz O, Whittle J, Harel D, Reggio G (eds.), Springer Berlin / Heidelberg, 2006; 321–335.
35. Jouault F, Tisi M. Towards incremental execution of atl transformations. *Theory and Practice of Model Transformations*, *Lecture Notes in Computer Science*, vol. 6142, Tratt L, Gogolla M (eds.), Springer Berlin / Heidelberg, 2010; 123–137.

36. Jouault F, Kurtev I. On the interoperability of model-to-model transformation languages. *Sci. Comput. Program.* October 2007; **68**:114–137, doi:10.1016/j.scico.2007.05.005. URL `http://portal.acm.org/citation.cfm?id=1290549.1298771`.
37. Wagelaar D, Tisi M, Cabot J, Jouault F. Towards a general composition semantics for rule-based model transformation. *Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, vol. 6981, Whittle J, Clark T, Khne T (eds.), Springer Berlin Heidelberg, 2011; 623–637, doi:10.1007/978-3-642-24485-8_46. URL `http://dx.doi.org/10.1007/978-3-642-24485-8_46`.
38. Jesús Sánchez Cuadrado EG Juan de Lara. The program is the model: Model transformations@run.time. *SLE*, LNCS, Springer, 2012.
39. Rose LM, *et al.*. A comparison of model migration tools. *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, 2010; 61–75.
40. Brent Boyer. Robust java benchmarking. http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html 2008.