Model Transformations with Tom

Jean-Christophe Bach[1], Xavier Crégut[2], Pierre-Etienne Moreau[3], and Marc Pantel[4]

INRIA & LORIA, Université de Lorraine[{1,3}]
Villers-lès-Nancy[{1,3}], France[{1,3}]

INPT-IRIT, Université de Toulouse[{2,4}]
Toulouse[{2,4}], France[{2,4}]

Abstract

Model Driven Engineering (MDE) advocates the use of Model Transformations (MT) in order to automate repetitive development tasks. Many different model transformation languages have been proposed with a significant tool development cost as common language elements like expressions, statements, ... must be built from scratch for each new language development tools. The Tom language is a shallow extension of Java tailored to describe and implement transformations of tree based data-structures. A key feature of Tom allows to map any Java data-structure to tree based data abstractions that can then be accessed by powerful non-linear, associative, commutative pattern matching. In this paper, we present how this approach can be used in order to develop model transformations, in particular relying on Eclipse Modeling Framework (EMF) based metamodeling facilities. This allows to provide a transformation language at a low cost both for the development of its tools and the training of its users.

1 Introduction

One of the key success of MDE comes from its ability to abstract complex problems and to provide a standard way to model them. For a given class of problems, the specification of the modeling language is called a *metamodel*, and for each specific problem, the abstract representation is called a *model* of the problem that conforms to its metamodel. There may exist several approaches to solve a problem, each of them generally consists in presenting the abstract problem in such a way that a specialized tool or solver can be applied. One of the main activity in MDE is thus to transform problems, *i.e.* models expressed in a given metamodel, into problems or views expressed in another representation, *i.e.* in another metamodel. The transformation of models is thus at the core of MDE applications.

The notion of metamodel has been formalized by the OMG[1] in the Meta Object

[1]Object Management Group, `http://www.omg.org`

Facility (MOF) standard[2] as a subset of the UML class diagram. Intuitively, a metamodel is composed of a set of *meta-classes* that contain *attributes* and *operations* like usual object oriented classes. The meta-classes can be linked by inheritance, and by *meta-relations*, either association or composition with an associated arity. Each model must conform to such a metamodel, *i.e.* it is a set of elements, valued element attributes and relations between elements conforming to their meta-definitions. A model transformation is a program that takes a model as input and returns a new model, possibly conforming to the same or another metamodel.

This principle is quite simple but, as usual in software engineering, expressing sophisticated model transformations in an executable formalism can be quite complex. There are at least two main approaches to describe a model transformation:

- either a model transformation is expressed as a sequence of elementary steps that build the target model step by step (instantiating new elements, assigning attributes, creating links, *etc.*) using information stored in the source model. This approach, usually called operational, is clearly imperative, as the target model is modified in each step. It can either be implemented using dedicated languages like Kermeta, QVT-Operational, ... or using reflexive libraries or generated code such as EMF [19] inside general purpose programming languages such as Java;
- or a model transformation can be defined as the relations that must exist between the source and the target models at the end of the transformation. This abstract presentation, sometimes called declarative or relational, is not directly executable, but, under some restriction, can be either translated to an operational transformation, or to a global function that takes in a step the whole source model as parameter and produces the whole target model as result (built for example with categorical pushout operators for graph rewriting based transformation languages).

Some authors [18] even consider a third architectural approach based on the introduction of an intermediate representation such as XML or any textual concrete syntax. The model transformation can then be described using a language such as XSLT or any language transformation toolset like Stratego/XT, ASF+SDF, Rascal, Spoofax, ...

Transformation language examples relying on both approaches will be presented later on in Section 5. However, the development and the use of new languages is costly as many common features such as expressions and statements must be implemented from scratch. Furthermore, users must learn how to benefit from the new capabilities provided by the languages. The MDE community is thus balancing between the development of new expressive languages and the use of libraries in general purpose programming languages. But the tools for dedicated languages (editors, interpreters, compilers, ...) hardly ever reach the quality of general purpose languages ones; and the use of libraries that allows to manipulate models in general purpose languages either through reflexion or code generation only provide the operational approach and their use in development is quite costly. In the reflexive approach, all the type verification is done at runtime as the elements name are handled as strings. The verification activities

[2] `http://www.omg.org/mof`

for transformations are thus very costly. In the code generation approach, the generated code usually provides static typing but very little model querying facilities. Thus the programming cost is quite heavy as the user must either add handwritten specific queries to ease the traversal of the metamodel, or do the explicit traversal each time a data is needed.

In order to bridge the gap between these two common approaches, we consider in the following the Tom language [12, 3], which lies halfway between them. On the one hand Tom is an extension of Java where every correct Java program is a correct Tom program. Therefore, in Tom it is possible to follow an operational approach and perform a model transformation using classical object oriented programming and EMF library. On the other hand, Tom adds declarative features to Java. An abstract term based representation can be associated to any data structure, even the most complex ones using a programmer provided, or automatically generated mapping. Sophisticated non linear associative commutative pattern matching and rule based programming can be used to define the transformations steps, without specifying in which order they are taken. Then, a specific construct called *strategy* is used to express the control and to specify how the transformation rules should be applied. All the constructs provided by Tom are then translated into pure Java relying on model management libraries like EMF. Thus, the end user can benefit from both classical programming, and model transformation technologies (operational and relational), without major efficiency penalty and without the additional cost of learning a completely new language. A third advantage of the proposed approach is that the language developers do not need to build a completely new language: they can rely on most existing Java tools.

In this paper, we present a technique to transform models with Tom/Java. It can be used for any model transformations as long as the metamodels can be expressed with the ECore formalism. This technique constitutes the first step towards a high-level extension of the Tom language. Section 2 introduces a practical example used in the next sections. The reader should note that the presented approach is general and is not restricted to the considered use case example; Section 3 presents the main constructs of the Tom language, tools developed to interface with EMF technology and a simple version of a model transformation; Section 4 explains how we implemented the use case in Tom and shows a generalized approach for writing models transformations with Tom; Section 5 summarizes existing model transformation languages, their advantages and drawbacks and how our proposal relates to them; Section 6 concludes and presents current and further works.

2 From Process models to Petri nets

In the following we rely on a case study introduced by Combemale *et al.* in [6] This example is both simple and rich enough to illustrate problems that may occur in the general case when considering more complex model transformations. It consists in transforming process described in the SimplePDL[3] formalism (Figure 1) into the Petri net formalism (Figure 2). This transformation may be used to verify properties on a process model, thanks to model-checkers based on Petri nets [7]. This verification

[3]Simple Process Description Language

aspect is not detailled in this paper that focuses on model transformation technologies.

Meta-models

The SimplePDL metamodel (Figure 1) defines the concept of *Process* composed of *ProcessElement*s. Each process element can be either a *WorkDefinition* or a *WorkSequence*. Work definitions are the activities that must be performed during a process. A work sequence defines a dependency relationship between two work definitions. The second work definition (*successor*) can be started — or finished — only when the first one (*predecessor*) is already started — or finished — according to the value of the attribute *linkType* (four possible values). Finally, a work definition can be defined as a nested process (*process* reference), allowing the definition of hierarchal processes.

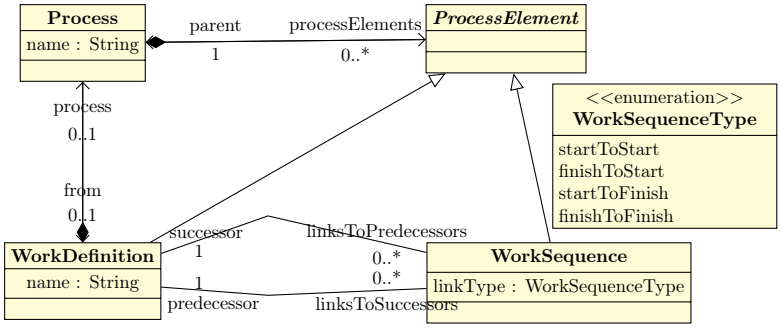

Figure 1: SimplePDL metamodel

The Petri net metamodel is shown on Figure 2. A *PetriNet* is composed of *Node*s that denote either a *Place* or a *Transition*. Nodes are linked by *Arc*. Arcs can be normal ones or read-arcs. An arc specifies the number of tokens (*weight*) consumed in the source place or produced in the target one when a transition is fired. A read-arc (second value in *ArcKind* enumeration) only checks tokens availability without removing them. A Petri net marking is defined by the number of tokens contained in each place (*marking*).

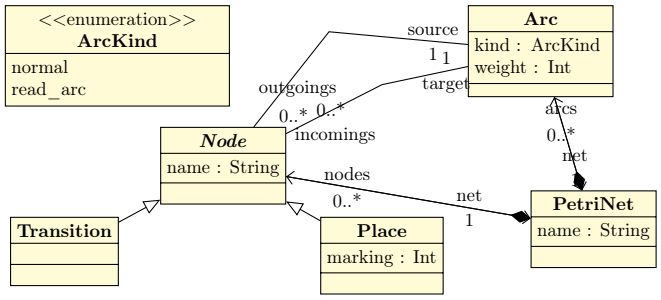

Figure 2: PetriNet metamodel

Example of transformation

The `root` process (Figure 3) is composed of two work definitions, `A` and `B` linked by a *start2start* work sequence, meaning that `B` can only start after `A` is started. `B` is itself described by a process (`child`) composed of two activities, `C` and `D` linked by a *finish2start* dependency. Thus, `C` has to be finished in order to start `D`.

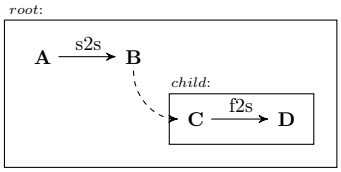

Figure 3: SimplePDL model

The transformation translates each process and work definition into a dedicated predefined Petri net template, and each work sequence into an arc. In a second step, when processes and work definitions are translated, arcs are generated to encode the synchronisation between the processes and their work definitions. A graphical representation is given below (Figure 4), where synchronisation is represented by dashed arcs, work sequences by thick green annotated arcs, places by red circles and transitions by blue squares. Details about elements composing this final Petri net are described in Section 4.

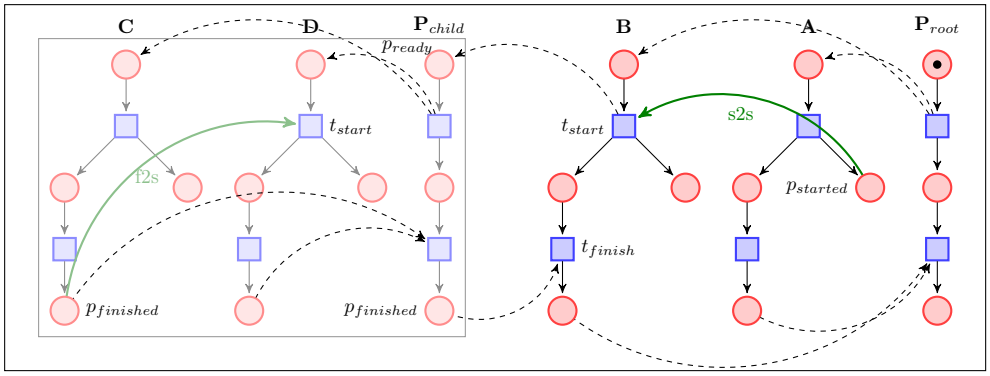

Figure 4: Complete Process described in the use case

3 Model transformation in Java with pattern-matching

To implement a model transformation such as the one sketched previously there are two main alternatives: either use a dedicated language (see Section 5), or use a general purpose language such as Java and associated model management libraries. Among the advantages of using Java we can mention efficiency, portability, and the fact that Java is a de facto industrial standard (well-known by engineers, integrated in existing processes, *etc.*). A main disadvantage is that Java is a low level, imperative language with respect to model transformations that is not well tailored for writing these kind of programs: there is no good support to query input models usually leading to complex code to retrieve a given piece of information and the declarative approach to model transformation cannot be expressed.

In the following we consider an extension of Java named Tom [3], whose goal is to make simpler the implementation of programs that manipulate tree-based data structures such as abstract syntax trees or XML documents for instance. The reader is invited to refer to `http://tom.loria.fr` for a more complete presentation.

Basically, Tom offers two new constructs: **%match**, a generalized *switch-case* construct that allows to discriminate upon objects instead of just plain data-types such as integers, and the ‘ (*backquote* construct) which simplifies the creation of objects.

Pattern matching

The **%match** statement is similar to a *switch-case* construct. It is composed of a list of *conditions-actions*: when a condition is satisfied, the corresponding action is fired. The main difference comes from the expressive power of the conditions: they are called *patterns* and correspond to trees that may contain variables. In the following we describe the transformation that associates a Petri net to a *Process* node of the input model:

```
1  public static void transformProcess(EObject subject, PetriNet pn) {
2    %match(subject) {
3      Process[name=n,from=src,processElements=pel] -> {
4        Node p_ready = ‘Place(n + "_p_ready", pn, ArcEList(), ArcEList(), 1);
5        Node t_start = ‘Transition(n + "_t_start", pn, ArcEList(), ArcEList());
6        ...
7        ‘Arc(t_start, p_ready, pn, normal(), 1);
8        /* creation of other Nodes and Arcs*/
9        ...
10       %match(pel) {
11         ProcessElementEList(_*,pe,_*) -> { transformProcessElement(‘pe, pn); }
12       }
13     }
14   }
15 }
```

Given a `subject`, the pattern `Process[name=n,from=src,processElements=pel]` at line 3 checks that the subject corresponds to a *Process* node. When it is the case, the variables `n`, `src`, and `pel` are initialized with the objects referenced by the attributes `name`, `from`, and `processElements`. In the general case, a pattern may contain nested patterns which add new constraints on the shape of sub-terms. Conjunctions (`&&`) and disjunctions (`||`) of patterns as well as multiple occurrences of a given variable (non linear patterns) may also be considered. The *action* part delimited by `-> { ... }` may contain Java statements but also Tom statements. For instance, line 4 combines a Java assignment and a Tom backquote construct which builds a *Place* (*i.e.* a Petri net node of the output model). The `‘Arc(...)` expression at line 7 updates the Petri net model `pn` by adding an *Arc* between the place `p_ready` and the transition `t_start`.

The last statement (from line 10 to 12) is interesting because it shows that nested **%match** constructs are allowed. It also illustrates the use of *list-matching*, also known as associative matching with neutral element. In this example, the pattern `ProcessElementEList(_*,pe,_*)` behaves like an iterator over `pel`: the action part is executed for each value of `pe` $\in$ `pel`.

The definition of the constructors `Process` and `ProcessElementEList` are not builtin in the language, they are derived from the metamodel given in Figure 1 relying on the EMF Java mapping. Thus, the syntax of patterns strongly depends on the considered metamodel.

Algebraic views

A **%match** construct depends on two different data-structures: the *subject* being matched (the subject is a reference to a plain Java object) and the *pattern* which is expressed in an algebraic language, namely Tom. In order to compile the pattern matching constructs, the Tom compiler needs to know the relationship between the implementation of objects (Java classes), and the algebraic view (pattern sorts and constructors). For this purpose, Tom offers an algebraic view definition formalism composed of two constructs: **%typeterm** and **%op**. The **%typeterm**, exemplified below, establishes a relation between the implementation data-type (`simplepdl.Process`) and the algebraic sort (`Process`):

```
1 %typeterm Process {
2   implement { simplepdl.Process }
3 }
```

The following **%op** constructs explains to the Tom compiler how a Java `Process` object can be viewed as an algebraic constructor, namely, `Process`:

```
1  %op Process Process(name:String, from:WorkDefinition, processElements:ProcessElementEList) {
2    is_fsym(t) { t instanceof simplepdl.Process }
3    get_slot(name, t) { ... }
4    get_slot(from, t) { ... }
5    get_slot(processElements, t) { ... }
6    make(name, from ,processElements) {
7      constructProcess(
8        SimplepdlFactory.eINSTANCE.create(SimplepdlPackage.eINSTANCE.getEClassifier("Process")),
9        new Object[]{ name, from, processElements })
10   }
11 }
```

The algebraic constructor `Process` has three arguments, respectively of sort `String`, `WorkDefinition`, and `ProcessElementElist`, and its codomain is the algebraic sort `Process` defined above. The construct **is_fsym** at line 2 is used by the pattern matching algorithm to check that the current constructor (*i.e.* `Process` in this example) is the root of the algebraic representation of the Java object. It is also needed for code generation. The Java code corresponds to the implementation of this predicate. Similarly, constructs **get_slot** (lines 3, 4 and 5) and **make** (line 6) define how to retrieve a given field in the data-structure, and how to build an instance of the considered constructor. This latter construct is used by the ‘ (backquote) construct to build terms.

For more details about the design of the Tom language, the reader is invited to refer to [12] and [3].

Towards an implementation of the transformation. Assuming that an algebraic view (*i.e.* a mapping) is defined for each element of the input and output metamodels, the considered model transformation, from SimplePDL to Petri nets, can be easily implemented using two passes over the input model: in a first pass, all *WorkDefinitions* are translated into Petri net templates, recursively calling `transformProcess` if necessary (introduced previously). In a second pass, the *WorkSequences* are translated into arcs according to *WorkSequenceType*. The resulting Petri net is a Tom term which can be used as any other term in a program. Alternately, it can be serialized by using EMF to have a standard model file usable by other tools.

This approach has a main drawback: it supposes that the transformation can be described in such a way that when building a link in the output model, the nodes it refers to have been generated in an earlier step. In the general case, this is not always possible, or this may imply an arbitrary large number of passes, which is not convenient nor efficient. A solution to this problem will be discussed and presented in Section 4.

In the following we present an automatic way to generate a *mapping* from a given metamodel.

Generator of algebraic views

EMF allows to generate Java code corresponding to a metamodel in a jar archive. We have designed a tool called Tom-EMF which uses reflection to load and inspect the classes contained in the jar archive. From theses classes, Tom-EMF can automatically retrieve the *EClassifier* (*EClass* and *EDataType*) and *EStructuralFeatures* in order to generate the **%typeterm** and **%op** constructs introduced above. For instance, when applying Tom-EMF on SimplePDL generated jar archive, the *Process* class produces a `%typeterm Process` and a `%op Process`.

When a reference indicates that an element may have multiple instances (for instance, Figure 1 shows a `processElements 0..*` relation which means that a `Process` can be composed of many `ProcessElements`), the tool generates list-matching operators such as `ProcessElementEList`, which are variadic, and whose matching is performed modulo associativity and neutral element.

Implementation design. The generator could have taken an `.ecore` file as input instead of a `.jar` file. The tool implementation would have been different, since we could not have used directly Java reflection. We would have used EMF mechanism to load ECore metamodels and replaced every use of classical Java reflection by EMF calls. We plan to implement this generator using Tom-EMF itself.

We made the choice to generate mappings without using a full EMF reflection in order to preserve static typing, and to avoid many dynamic casts (and thus to be more efficient and to avoid potential runtime errors). A drawback is that Java files and Tom mappings have to be generated before any use (this adds an extra generation step using Eclipse).

Changes in EMF specification would not have any consequences on Tom language and the Tom compiler, but it may have an impact on the Tom-EMF mapping generator. In practice, this is not a big issue as EMF is based on the MOF standard.

4 A generalized approach for model transformations using strategies

In the previous section we have shown how pattern matching capabilities offered by Tom can be used in conjunction with Java classes generated by EMF. The generation of mappings being automatic, this provides a quite simple framework that can be used to define model transformations.

However, a main drawback is the need to take care of the order in which transformation steps have to be performed, specially when the transformation has to reference elements not yet created or completed in the output model.

In the following we present a generalized two-steps approach where we separate the notion of transformation from the control of these transformations. In a first step we specify in a declarative way how each specific sub-part of the input model should be transformed. Each transformation is applied separately, leading to a model where the various sub-parts are not connected: they contain nodes, called *resolve nodes*, which describe the *intension* of being connected to another sub-part when it will become available. This approach is strongly inspired by the *resolve* constructs of ATL [9] and QVT [14].

In a second step, all the *resolve nodes* are traversed and replaced by *links*, to build the final output model.

Generic traversal strategies

In addition to ‘ (*backquote*) and **%match** constructs, Tom offers a third construct, **%strategy**, which encodes the notion of elementary transformation rule. For instance, let us consider the following snippet of code:

```
1 %strategy Process2PetriNet() extends Identity() {
2   visit Process {
3     Process[name=name,from=from] -> { <Host code + Tom code block> }
4   }
5 }
6 ...
7 ‘TopDown(Process2PetriNet()).visit(root_process);
8 ...
```

This defines an elementary transformation called `Process2PetriNet` whose default behavior is the `Identity`, meaning that no transformation occurs when the considered rule cannot be applied[4]. The `visit Process` constructs is a first filter which specifies the sort of objects (`Process` in this case) on which the rule should be applied. Then a classical Tom-rule composed of a pattern and an action is defined.

The main particularity of a **%strategy** construct is that it is not automatically fired. Its application should be controlled by a *strategy*. For instance, the `TopDown(Process2PetriNet())` expression means that the rule `Process2PetriNet()` is applied in a *top-down* way on the term `root_process`. Tom offers several primitive strategies, such as `Identity`, `Fail`, `One`, `All`, `Choice` and `Sequence`, which can be combined, even recursively, to build more powerful strategies such as `Repeat`, `TopDown`, or `Innermost` for instance.
For more precise information about strategies, the reader can refer to [4] and to the dedicated page[5] on Tom project website.

Decomposition of the transformation

To address the issues mentioned at the beginning of this section, we propose an approach where a model transformation is specified in terms of elementary transformations which

[4]This is in contrast to `Fail` which specifies that the transformation *fails* when the rule cannot be applied.

[5]`http://tom.loria.fr/wiki/index.php5/Documentation:Strategies`

are parts of the whole transformation. Each elementary transformation being implemented by an elementary strategy. In this approach, the order of application of the transformations is not defined in the transformation itself, therefore, when an elementary transformation has to reference an element which is not yet created or completed in the output model (*e.g.* elements which should be created in another elementary transformation), we have to create a temporary object called *resolve object*, in reference to the *resolveTemp* construct of ATL [9] (see also *resolveIn* in QVT [14]). Each *resolve object* is stored in a two level hash-map which maps each source element we are processing (`WorkDefinition` in our use case) to another hash-map. This latter map connects a name (a label such as `"p_ready"` for example) to a target element which just has been created (the place referenced by the variable `p_ready` in our example).

Once each atomic transformation has been applied, the result is composed of temporary resolve objects and partial results which need to be reconnected. Thanks to the table, *resolve objects* can be replaced by target objects in corresponding partial results, and we obtain the final result of the transformation.

Elementary transformations. In our use case, we consider three atomic transformations: `Process2PetriNet`, `WorkDefinition2PetriNet`, and `WorkSequence2PetriNet`). Each of them is implemented by a **%strategy** construct.

The first one, `Process2PetriNet`, creates the Petri net that corresponds to the image of a `Process`. This Petri net is composed of three places (p_{ready}, $p_{running}$ and $p_{finished}$), two transitions (t_{start} and t_{finish}) and four arcs. During the transformation, each created element (*i.e.* transitions and places) is stored in a hash-map (`map`), which is itself associated to the process being transformed (`p` is a variable assigned to the process which is matched). The last part of the strategy creates *resolve objects*: they encode the link that should be created when all parts of the model are available. In our running example, the *resolve object* is a reference to the transition t_{start}, resulting of the transformation of `WorkDefinition`, that will be connected via an arc to the place p_{ready} of the current generated Petri net (see Figure 4 to get the "big picture"). Figure 4 shows the resulting Petri net of `Process2PetriNet` transformation (dashed nodes are *resolve* nodes). Figure 4 sketches the implementation:

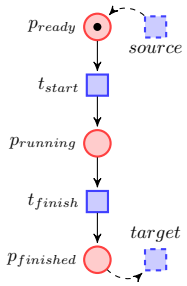

Figure 5: Petri net template for a Process

```
1  %strategy Process2PetriNet(pn:PetriNet,table:HashMap) extends Identity() {
2    visit Process {
3      p@Process[name=name,from=from] -> {
4        Node p_ready = `Place(name + "_ready", pn, ArcEList(), ArcEList(), 1);
5        Node t_start = `Transition(name + "_start", pn,ArcEList(), ArcEList());
6        ... // creation of p_running, t_finish, and p_finished
7        `Arc(t_start, p_ready, pn,normal());
8        ... // creation of the 3 remainings arcs
9        HashMap map = new HashMap();
10       map.put("p_ready", p_ready);
11       ... // store p_running, p_finished, t_start, and t_finish
12       table.put(`p, map);
13       // `from is the WorkDefinition which contains the current Process, in case of hierarchical description
14       if (`from!=null) {
15         Transition source = new ResolveWorkDefinitionTransition(`from, "t_start");
```

```
16          ‘Arc(p_ready, source, pn, normal());
17          Transition target = new ResolveWorkDefinitionTransition(‘from, "t_finish");
18          ‘Arc(target, p_finished, pn, read_arc());
19        }
20      }
21    }
22  }
```

Figure 6: Process2PetriNet strategy

The two other atomic transformations implemented by `WorkDefinition2PetriNet` and `WorkSequence2PetriNet` strategies are based on the same principle, we just describe them without giving their corresponding code.

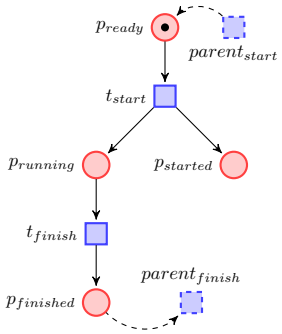

Figure 7: Petri net template for a WorkDefinition

The `WorkDefinition2PetriNet` strategy creates all Petri net elements which define the image *WorkDefinition*. This Petri net is composed of four places (p_{ready}, $p_{running}$, $p_{started}$ and $p_{finished}$), two transitions (t_{start} and t_{finish}), and five arcs (Figure 6). The only difference between a process representation and a work definition is the fact that there is an additional place $p_{started}$ after the t_{start} transition. The last part of the strategy creates *resolve objects* representing nodes of the parent *Process* to which the *WorkDefinition* is connected.

A work sequence between two work definitions is simply represented by a read-arc between two activities (Figure 6). A read-arc controls that a transition is enabled (it checks if a token is present without removing it). Therefore, the `WorkSequence2PetriNet` strategy consists in creating an *Arc* whose source and target are both *resolve nodes*.

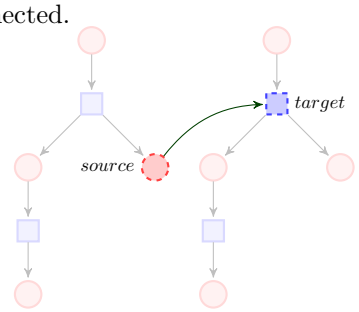

Figure 8: Petri net template a WorkSequence

Controlling application of transformations. We have described three elementary transformations using strategies. The order in which the transformations should be applied is not encoded in the transformations themselves. Indeed, the order is not relevant and the three transformations can be applied in any order. In Figure 6 we give a possible strategy: first we transform a `WorkSequence` into arcs, and then we transform `Process` and `WorkDefinition`. This order has been chosen arbitrarily and could be different without any consequences.

```
1 public static void main(String[] args) {
2  ...
3 // transformer is a composition of all atomic transformations
4 Strategy transformer = ‘Sequence(TopDown(WorkSequence2PetriNet(pn)),
5                                  TopDown(Process2PetriNet(pn)),
6                                  TopDown(WorkDefinition2PetriNet(pn)));
7 // call of transformer on the root process
```

```
8 transformer.visit(p_root);
9 ...
```

Figure 9: Composition of builtin and custom strategies

Connecting intermediate results. At this stage of the presentation we have shown how a complex model transformation can be described by several independent transformations, encoded by elementary strategies. To achieve this goal we have introduced temporary *resolve nodes* that should be eliminated and replaced by corresponding *links* in the resulting model.

Since implementations of models are statically typed (thanks to EMF) we had to introduce one type of *resolve object* per type. For example, to implement a *resolve node* that corresponds to a place in a `WorkDefinition`, we consider the following class:

```
1 private static class ResolveWorkDefinitionPlace extends petrinet.Place {
2   public String name;
3   public simplepdl.WorkDefinition o;
4   public ResolveWorkDefinitionPlace(simplepdl.WorkDefinition o, String name) {
5     this.name = name;
6     this.o = o;
7   }
8 }
```

Let us now consider a simple PDL process composed by two `WorkDefinition` and a `WorkSequence`. Applying the `transformer` strategy leads to four unconnected Petri nets, as illustrated (Figure 9). In Figure 9 we introduce a meta-strategy named `Resolve`, whose goal is to replace all *resolve objects* by the corresponding image, which are stored in the two-level hash-map. The strategy *Resolve* is applied in a top-down way on the resulting unconnected Petri nets that the elementary transformations produced. As a result, every *Node* is visited and each `ResolveWorkDefinition` node for instance is replaced by the node stored in the `table`.

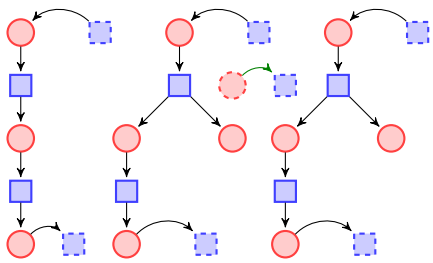

Figure 10: "Unresolved" Petri net

```
1  %strategy Resolve(translator:SimplePDLToPetri) extends Identity() {
2    visit Place {
3      ResolveWorkDefinitionPlace[o=o,name=name] -> {
4        Place res = (Place) translator.table.get(`o).get(`name);
5        return res;
6      }
7    }
8    visit Transition {
9      ResolveWorkDefinitionTransition[o=o,name=name] -> {
10       return (Transition) translator.table.get(`o).get(`name);
11     }
12     ResolveProcessTransition[o=o,name=name] -> {
13       return (Transition) translator.table.get(`o).get(`name);
14     }
15   }
16 }
```

Figure 11: Resolve strategy

In the current implementation, the strategy `Resolve` should be manually coded by the user. We are currently experimenting an extension of the proposed approach where the `Resolve` strategy would be automatically inferred and generated from the elementary transformations. This is discussed in Section 6.

5 Related work

This part focuses on full blown languages and experiments that have been applied on real case studies and have been available for several years either in the academic or industrial worlds.

Many languages have been designed in order to ease the writing of model transformations using model manipulation operators. These languages allow to access model contents using either graph pattern matching or object query languages usually derived from OCL. One of the differences of our approach is that instead of designing a whole new language to transform models, we use an efficient language relying on strong theoretical basis and designed to be used with general purpose languages.

The Object Management Group (OMG) has defined the Query View Transformation standard (QVT) to provide model transformation technologies for the Model Driven Architecture (MDA). Modeling languages are defined using MOF and are manipulated using OCL. The QVT standard proposes both the relational and operational approaches. The main difference is that operational ones require to describe the control as part of the transformation whereas it is handled by the execution engine for relational ones. However, the current implementations of the standard usually restrict themselves to a single one. In this paper we focus on implementation of the standard in the Eclipse world. The ATL language has been designed during the standardization process of QVT [9]. It does not rely on the QVT concrete syntax but implements most of the operational and part of the relational approaches. Medini-QVT-Relational[6] implements the relational part of the standard whereas the Eclipse M2M QVT-Operational[7] implements the operational part of the standard. The relational approach consists in defining rules that map target elements and source elements. When the transformation is applied, the execution engine matches rules according to the source model and run them in the right order to avoid dangling references to not yet initialized target elements. Tom's strategies corresponds to rules and the order of rule application is defined by meta-strategies (and thus should be, at the moment, defined by the programmer).

Several languages that follow the operational approach without implementing the QVT standards are available in the EMF world. Kermeta[8] allows to implement methods in meta-classes [13]. Model transformations are thus expressed as methods defined on the meta-classes. In order to avoid polluting the metamodels, Kermeta allows to extend meta-classes using aspect technologies. The *Epsilon Transformation Language* (ETL) [11] relies on a core common language that is used in many different tools in the Epsilon[9] toolbox. The XTend[10] language follows a similar approach. In operational

[6] `http://projects.ikv.de/qvt/`
[7] `http://www.eclipse.org/m2m/`
[8] `http://www.kermeta.org/`
[9] `http://www.eclipse.org/gmt/epsilon/`
[10] `http://www.eclipse.org/Xtext/#xtend2`

approaches, programmers have full control on the transformation but have to deal with the mapping between source and target elements, dangling references, etc. Tom's meta-strategies allow to separate the construction of parts of the target model (often described as functions in operational approach) and the sequencing of theses construction.

In the same spirit as Tom, term rewriting tools can be adapted in order to implement model transformations. Several experiments with various encodings of models have been conducted and led to some methods for using existing languages or to new toolsets. The Maude language is based on term equational rewriting systems. It provides object oriented facilities that can be used to implement metamodels and models. Its use has been experimented by several research teams [5, 16, 15] and implemented for example in the Moment[11] project.

Term rewriting has been used for the last 30 years for implemented program transformations. ASF+SDF, Stratego/XT, Elan and Tom have been available for a long time and applied on real world case studies. The main issue in order to use them for model transformation is to switch from terms to graphs. The Spoofax[12] toolset is based on Stratego/XT [10, 8].

Graph rewriting for model transformations has been experimented in the last 20 years [20, 17] using various categorical encodings like single and double pushout and specification formalism like graph types or triple graph grammars. Many tools have been implemented. The following ones are currently available in the Eclipse world. The Moflon[13] toolset relies on triple graph grammar in order to implement model transformations [1]. The Henshin[14] project is the followup of many experiments [2] in Eclipse based on the AGG toolset. The main issue of graph rewriting is that it relies on quite costly synthesis technologies in order to build the transformation function from the elementary graph rewriting rules and the scalability of the categorical technologies is still to be experimented.

In the same spirit as the Tom shallow language extension approach, MPS[15], the Meta Programming System, relies on lightweight language extensions that are translated to the core language, currently Java. MPS provides an integrated tool for the definition of languages extensions. It could thus be used for Tom implementation. But, it does not provide currently model transformation extension to Java. Tom also relies on state of the art term rewriting technologies and especially efficient term management and pattern matching.

6 Conclusion and future work

In this paper we have presented the Tom programming language and we have showed how pattern matching and algebraic views can be used to encode model transformations in a more abstract way than in pure Java, using EMF.

In a second part, we have introduced the notion of strategy, and we have showed how they can be used to encode elementary transformations, where the notion of scheduling

[11] http://www.cs.le.ac.uk/people/aboronat/tools/moment2/
[12] http://strategoxt.org/Spoofax
[13] http://www.moflon.org/
[14] http://www.eclipse.org/modeling/emft/henshin/
[15] http://www.jetbrains.com/mps/

is no longer part of the transformation itself.

To achieve this result, we have introduced intermediate QVT derived *resolve objects* and a two-level hash-map that stores them. Then, we have showed how they can be replaced by *links* of the model, using an elegant `TopDown(Resolve())` meta-strategy.

What we have presented is a first step towards a high-level language integrated into Java. The method we presented can be used to specify any model transformation between any two different models. Our next objective is to introduce a higher-level construct **%transformation** which automates the generation of most of the code presented previously: elementary strategies, *resolve objects* classes, and the `Resolve` meta-strategy can be automatically generated.

This new construct will be composed of model transformation rules, and each rule would be compiled as a strategy. A developer will only have to write the transformation itself without having to take care of *resolve objects*, the two-level hash-map, as well as the `Resolve` strategy. All the needed information will be encoded in the transformation, using a domain specific syntax dedicated to the transformation of models. The goal is to simplify the writing of models transformations in the Java world.

The second part of incoming work is the extension of the strategy language to offer the possibility to create parametrized strategies. Parameter could be the type of link the transformation developer wants to follow. It would make it more flexible.

A third part of future work will be the extension of the mapping generator itself: for the moment, it is EMF-based, but we could generalize it to handle other technologies.

Once Tom language will be extended, we will be able to express complex transformations in Java in an easy way. Then we will be able to build complete execution traces to verify models transformations. With those traces, it will be possible to reconstruct the transformations chain to point a potential problem (for example, given by the results of a model-checker).

References

[1] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. Moflon: A standard-compliant metamodeling framework with graph transformations. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2006.

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.

[3] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: piggybacking rewriting on java. In *Proceedings of the 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Rewriting strategies in java. *Electr. Notes Theor. Comput. Sci.*, 219:97–111, 2008.

[5] Artur Boronat and José Meseguer. Moment2: Emf model transformations in maude. In Antonio Vallecillo and Goiuria Sagardui, editors, *JISBD*, pages 178–179, 2009.

[6] Benoît Combemale, Pierre-Loïc Garoche, Xavier Crégut, Xavier Thirioux, and François Vernadat. Towards a Formal Verification of Process Model's Properties SimplePDL and TOCL Case Study. In *ICEIS (3)*, pages 80–89, 2007.

[7] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(6), 2009.

[8] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software and System Modeling*, 9(3):375–402, 2010.

[9] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.

[10] Lennart C. L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *OOPSLA*, pages 444–463. ACM, 2010.

[11] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2008.

[12] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer Berlin / Heidelberg, 2003.

[13] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[14] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation (QVT) Specification, version 1.0*, April 2008.

[15] José Raúl Romero, José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9):187–207, 2007.

[16] Vlad Rusu. Embedding domain-specific modelling languages in maude specifications. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[17] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008.

[18] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[19] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[20] Gabriele Taentzer. What algebraic graph transformations can do for model transformations. *ECEASST*, 30, 2010.