# Matching Model-Snippets

Rodrigo Ramos[1,2], Olivier Barais[2], and Jean-Marc Jézéquel[2]

[1] Centre of Informatics, Federal University of Pernambuco
P.O. Box 7851, CEP 50732970, Recife, Brazil
[2] IRISA / INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

**Abstract.** An important demand in Model-Driven Development is the simple and efficient expression of model patterns. Current approaches tend to distinguish the language they use to express patterns from the one for modelling. Consequently, productivity is reduced by dealing with a distinct new language, and new intermediate steps are introduced in order to support pattern-matching. In this paper we propose a framework for expressing patterns as *model-snippets*. We present how model-snippets are specified upon concepts in a given domain (meta-model), and how we perform pattern-matching with model-snippets, whatever the meta-model. We also provide an implementation which is well integrated with existing technologies, such as Eclipse Modelling Framework.

## 1 Introduction

Recently, hopes that modelling might play a more important role in the software engineering process have been lived up by Model Driven Development initiatives. These initiatives have been mainly advanced by the Object Management Group and IBM through the Eclipse foundation. The main proposition of this approach consists of considering models as first-class artefacts within the software development process. The recent focus on these assets has raised several new issues [1]. An important demand is the introduction of more automatic ways for searching into model repositories. This demand can be rephrased into the ability to perform efficient pattern matching at the model level.

Pattern-matching at the design level is also used, for example, in order to perform declarative transformations [2], to recover design-patterns in an object-oriented software design [3], or to identify join points in a model from a pointcut expression to weave aspects into models [4,5]. In all these cases, we can imagine patterns as *model snippets* with some conditions, which are possibly expressed by some predicates over the models that they wish to match. In fact, several languages have been proposed to express these patterns [4,5] in this way. Unfortunately, most of these languages were designed to a specific domain, defined by a specific meta-model. Moreover, approaches that implement pattern matching over these languages tend to also be specific to these meta-models. This limitation has been obstructing the application of pattern matching over new meta-models, since each of them requires new support for pattern matching. For instance, despite of the good results of current methods for aspect-oriented

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 121–135, 2007.

modelling, it is still difficult to play with its concepts over non object-oriented paradigms or, even more, meta-models that do not extend UML.

In this paper, we propose a generic pattern framework for expressing pattern at the model level and for performing pattern-matching. The language for expressing patterns is built on demand, conforming to an input meta-model of the model that we wish to match. The main goal of simplifying the expression of patterns at the model level, avoiding any textual regular expressions, is to assist the design of these patterns with existing model editors for the user, whatever the meta-model. Moreover, we consistently define model patterns upon the concepts of *model-typing* [6], which permits us to better manage such patterns during model evolution. The results of this work are also integrated with existing technologies, such as EMF[1] (Eclipse Modelling Framework), Kermeta[1] (a meta-model engineering environment within Eclipse) and Flora-2[1] (An object-oriented knowledge base language that extends Prolog).

The next section complements this introduction with a motivating example, which will also guild us throughout the paper. Then, Section 2 presents our approach for *meta pattern-matching* in details, and Section 3 describes how this is implemented and integrated with Kermeta and Flora-2. Finally, Section 4 relates this work with existing approaches, and Section 5 concludes this paper.

### 1.1 Motivating Example

In order to motivate this work, a small state-machine meta-model is presented in this section, in which we intend to perform a pattern-matching. As any typical state-machine, it is mainly formed by a set of states and transitions among the states. Additionally, the meta-model might include some constraints (expressed in OCL, for example), which among other things may specify that the state-machine must have a unique initial state.

Now, assume that we have a state-machine obeying this meta-model (left-hand side of Fig. 1), which is composed by several complex states and transitions and detailed with actions to be executed during the state-machine life-cycle.

Moreover, suppose that we want to find some simple patterns over this model, such as cyclic-transitions or, in the opposite way, any transition that links two distinct states. By the simple nature of these patterns, we wish to avoid dealing with complex details of the state-machine meta-model, specifying just what is needed for performing pattern-matching. Moreover, it makes sense to express patterns in a similar language in order to make their specification easier and avoid learning new languages. Another concern is to use the same mechanism for each meta-model, instead of recreate them to each new meta-model. We can summarise these concerns, as follows:

- *How to have a simple way to express a valid pattern?*
- *How can we be guided by our meta-model in the specification of a pattern?*
- *Can we express patterns using existing editors for our meta-model?*
- *Are our models already supported by any existing pattern-matching tool?*

[1] see www.eclipse.org/emf, www.kermeta.org and flora.sourceforge.net

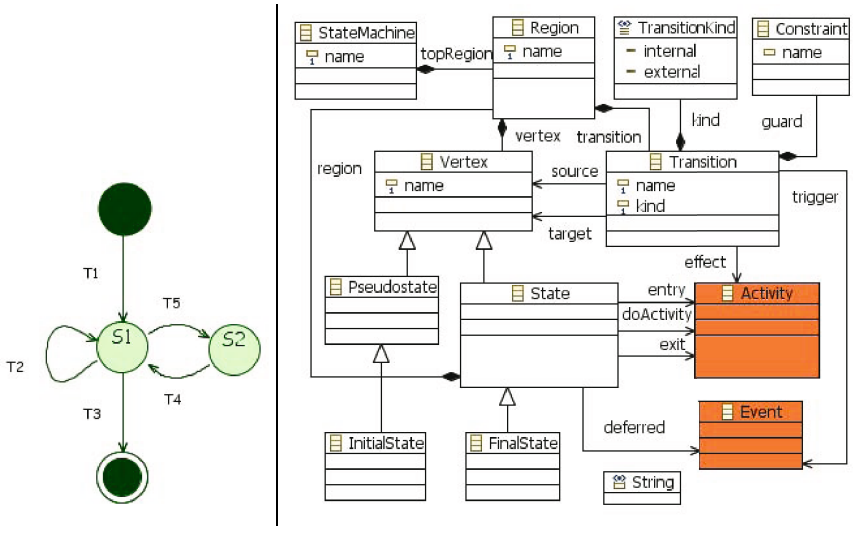


**Fig. 1.** A model and a meta-model of state-machine

All things considered, we have identified, as other authors [4], that probably the simplest way to express patterns is using the same concepts that we find in our meta-model, i.e. expressing patterns as *snippets* of a valid model in our meta-model. In order to realise this idea, and also to support its application in new meta-models, we discuss throughout this paper a pattern framework for expressing pattern as model snippets and for performing pattern-matching. The questions raised in this section are also addressed during the paper, using the small example of the state-machine to illustrate our discussion.

## 2 Patterns as Model-Snippets

As suggested in the previous section, patterns might be expressed as *model snippets*, such as the UML templates [4] in Fig. 2. Having patterns expressed in this way, it is possible to allow a user to draw patterns using editors that he is used to, when drawing the models that he intents to match.

Each model-snippet defines a set of information existing in the model that we wish to match. For example, in Fig. 2, a class named *Trace* is declared with

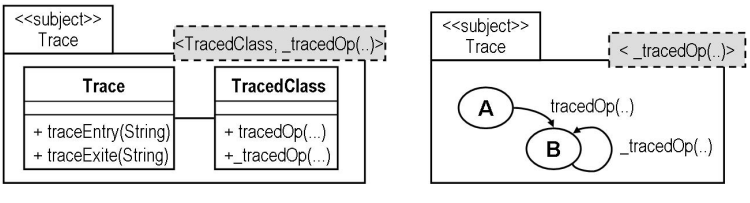


**Fig. 2.** UML Templates of a Class and State Diagram

two methods (*traceEntry* and *traceExit*). Whenever, a class contains the same name and methods, it matches with *Trace*. Obviously, the matched class might have also more information, such as methods, attributes or associations.

Observe that the snippet in Fig. 2 was constructed for UML models [4]. Many other domain specific languages could take advantage of a similar approach. In brief, we can define *model snippets* as:

> A set of objects S is a ***model snippet*** of meta-model $MM$ iff:
> – every object in S is an instance of a metaclass defined in MM;
> – there exists a set M where M is a Valid Model w.r.t. MM and S is subset or equal to M.
>
> where, we assume that a model and a meta-model are respectively sets of EMOF objects and classes.

Every valid model is also a snippet (w.r.t its own metamodel) and so is every model that can be obtained by removing objects from that model. But not every model that may be obtained by adding objects to such a model.

With this in mind, we show how we can express model-snippets in any domain. We present in Sect. 2.1 a pattern-framework with the minimal elements that forms a pattern. In Sect. 2.2, we show how to create *model snippets* and to customise this framework according to a target meta-model. Finally, on Sect. 2.3 we formally define what we mean by pattern-matching using model-snippets.

### 2.1 Pattern-Framework Meta-model

Taking a closer look at the model-snippet in Fig. 2, we can see the snippet as an instance of the UML meta-model. All elements in the snippets are instances of a UML classifier, and furthermore inherit from a superclass *NamedElement*. For instance, *Trace* is an instance of a UML *Class* identified with a feature *name* equal to $'Trace'$, and the method *traceEntry* is an instance of *Operator* with *name* equal to $'traceEntry'$. The same happens in the model that we want to match. An important finding from this observation is that a snippet specifies a subset of instances, and of associations among them, in the model that we want to match. This set of instances is the information that we use to match models. However, a pattern seems to be a little more complex than just a set of instances of a meta-model.

It is clear from Fig. 2 that a pattern is mainly formed by a *snippet* part (in each package of the figure) and a sequence of free-variables over this *snippet* part (in rectangle on the top-right side of each package). The purpose of variables is to define the selection criteria for a particular model element. A variant can also be conceptually seen as a placeholder for any element in the *intended model* that is matched to it. In most cases, variables represent elements that play a significant role in the pattern, and that we have a special interest in matching with. Contrary to variables, non-variables must be directly associated to a unique element in the *intended model*, containing all features which identify the element.

Nearly all meta-models define a special feature that uniquely identifies each element of their models. As we wish to be able to match variables with more

than one element in the model, we do not take into account this identifier during pattern-matching of variables. For instance, *TracedClass* is a variable in Fig. 2. So it matches to any class, with any name, that has the same methods than *TracedClass* and an association to a class named *'Trace'*. *Trace* is a non-variable, and, furthermore, we take into account its *name* during pattern-matching. As in most of the cases, UML uses a feature *name* as an identifier. However, the feature can change in other meta-models.

The more information is expressed in the structural part of the pattern, the more precise is the pattern-matching. However, an excessive and detailed *snippet* might also uncover all positive matches. For this reason, as any model, a pattern can have additional constraints, which help to better describe the pattern and to relate variables with other elements in the model.

Note that constraints between variables and non-variables in a pattern should still be valid after they have been matched with elements in the *intended model*. From this standpoint, constraints might also help to describe false positives in the set of possible matchings. In doing so, constraints can represent *Negative Application Conditions* (NAC) in a pattern-matching, improving the accuracy of the matching process.

Based on the concepts presented above, a possible generic meta-model for patterns could be expressed as the one in Fig. 3. In the meta-model, *Pattern* represents the whole pattern, and *PatternStructure* represents its structure. The structural part contains a *PModel* with a set of instances of classes in a given meta-model, which we wish to be related to the metamodel that describes the models in which we look for matches (*intended model*). *PatternStructure* has also a set of *Role*, which express pattern variables in the structural part. Additionally, the pattern can also have some constraints, or *invariants*, that, here, might be expressed in OCL or Kermeta.

Fig. 3 presents what we call a *Pattern Framework*. *PModel* is the point (*hot spots*) where the framework can be adapted or specialised by the developer. The specialisation of our framework to the meta-model that describes the *intended models* is described in the next section.

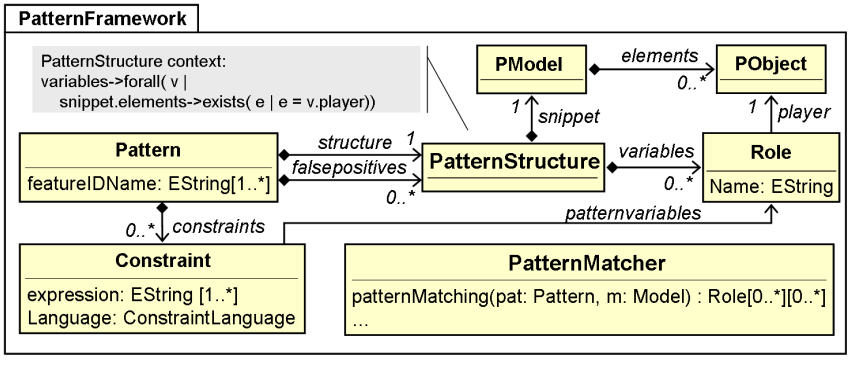


**Fig. 3.** Meta-model of the pattern framework

### 2.2 Constructing Model-Snippets

Most of the time, the meta-model ($MM$) of the *intended model* is too restrictive to represent patterns. The reason for that is very simple; patterns have to be expressed in a higher level of abstraction, such as *model-snippets* (see definition in the beginning of Sect. 2). For example in the state machine meta-model, it is totally understandable that someone does not want to provide the mandatory *event* of a $Transition$, or even want to instantiate a $Vertex$, which is defined as abstract, in order to match over instances of $State$ or $PseudoState$.

We wish that a snippet relies as much as possible on the same concepts of $MM$. In order to do that, we construct by demand a more flexible meta-model ($MM'$) that allows us to represent abstract patterns with all concepts of the meta-model of the *intended model*. If we imagine a flexible meta-model $MM'$ that is equals to $MM$, except that:

- No invariant or pre-condition is defined in $MM'$;
- All features of all classes in $MM'$ are optional;
- $MM'$ has no abstract element.

Then, we can notice that all concepts in $MM$ are also represented in $MM'$. $MM'$ describe a wider range of models, including all models described by $MM$ (see Fig. 4). This is obtained by removing all restrictions that exist in $MM$: invariants, mandatory features, nonexistence of instances of certain class. To allow features to be optional, we just set its lower bound as zero. These restrictions can be taken rewritten as invariants over a group of classes $S$, and any group of classes with a weaker invariant could be taken as a generalisation of $S$ [6].

Note that any model that conforms to $MM$ also conforms to $MM'$, and, furthermore, any model-snippet that conforms to $MM$ also conforms to $MM'$. A detailed discussion about this conformance is presented in Sect. 2.4. This is an important result, it shows that we can still use existing graphical editors to draw pattern snippets and use them for pattern-matching. It also says that any meta-model that generalises $MM$ can be used to specify more abstract patterns.

For example, we can generate a new and flexible meta-model ($MM'$) from the state machine meta-model in Fig. 1 ($MM$). $MM'$ might describe, for instance, a *Region* with zero *InitialStates* or a *State* with no *Activity*. It also describes $Vertex$ as a concrete class, allowing instances of it.

In order to finish the automatic building the meta-model of model snippets according to a specific meta-model, we need to merge our general framework for patterns (Fig. 3) with this restrictiveness meta-model ($MM'$), which takes

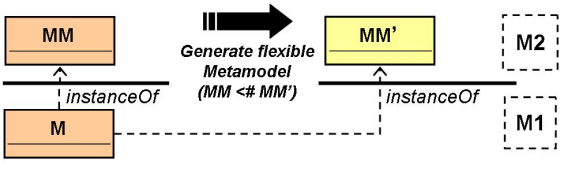


**Fig. 4.** Process for deriving a meta-model for pattern *snippets*

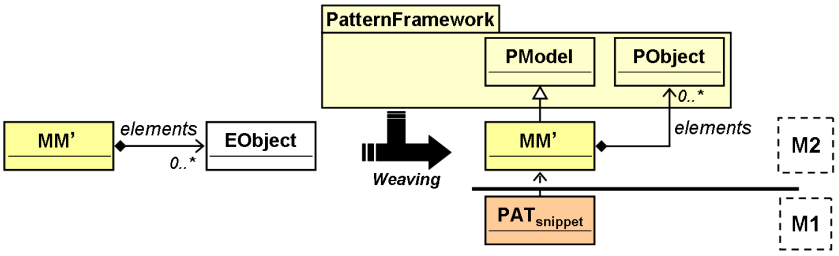


**Fig. 5.** Process for customising the pattern framework

into account concepts in the intended domain ($MM$). This composition is called a weaving because this transformation integrates $PObject$, which comes from the pattern framework ($PatternFramework$), as a superclass of all meta-classes in $MM'$. The transformation can be compared to an interface introduction in AspectJ[1] that adds a new superclass to a type. Our weaving process is equivalent to this mechanism. The implicit pointcut used in our weaving applies the introduction of the $PObject$ superclass into all the metaclasses that do not have any superclass. The whole process to derive $MM'$ from $MM$ and to permit the specification of valid pattern $snippet(PAT_{snippet})$ is presented in Fig. 5.

For example, we can generate a new meta-model that weaves our *pattern framework* and the state machine meta-model. It includes classes from both framework and meta-model, and additionally hook all classes from the latter with $PObject$. Then, all classes that do not have a super class in $MM'$ inherit from $PObject$ after the weaving process. Fig. 6 shows the inheritance between some of these classes and $PObject$. As a result, we obtain a meta-model that can be used to express model-snippets and also can be taken as an input of the pattern-matching mechanism.

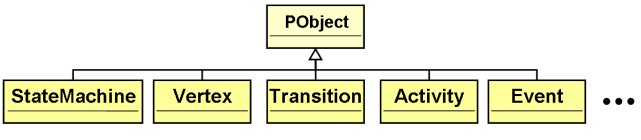


**Fig. 6.** Weaved state machine meta-model

### 2.3 Pattern-Matching Behaviour

In this section, we describe what we mean by pattern-matching using *model snippets*. We make use of a formal notation that looks like OCL, but with explicit use of quantifier operators of the first order logic. We also assume that any instance of $MM'$ and $MM$ is also an instance of $Model$, as it is presented in Fig. 7. Where, $Model$ contains a *elements* reference with a set of EMOF object instances.

[1] see www.eclipse.org/aspectj.

In order to better represent our ideas, we define some auxiliary (private) methods in $PatternMatcher$. We have included the method $Booleancontains(Obj_1 : EObject, Obj_2 : EObject)$ to indicate when an object $Obj_1$ contains all the features that an object $Obj_2$ has, and $Booleancontains(M_1 : Model, M_2 : Model)$ to indicate when all objects in model $M1$ have all objects in a model $M2$. These methods are specially used to know when a model $M_2$ is a snippet of $M_1$.

$$contains(M_1 : Model, M_2 : Model) = contains(M_1, M_2, \{\}, \{\}) \quad (1)$$
$$contains(Obj_1 : EObject, Obj_2 : EObject) = contains(Obj_1, Obj_2, \{\}, \{\}) \quad (2)$$
$$contains(M_1 : Model, M_2 : Model, fIds : EString[0..*], roles : Role[0..*]) = \quad (3)$$
$$\forall e_2 \in M_2.elements \bullet \exists e_1 | contains(e_1, e_2, fIds, roles) \quad (4)$$
$$contains(Obj_1 : EObject, Obj_2 : EObject, fIds : EString[0..*], \quad (5)$$
$$roles : Role[0..*]) = \quad (6)$$
$$\forall r_2 \in Obj_2.getAllReferences() \mid Obj_2.eGet(r_2).isDefined() \bullet \quad (7)$$
$$\exists r_1 :\in Obj_1.getAllReferences() | r_2.name = r_1.name \wedge \quad (8)$$
$$contains(Obj_1.eGet(r_1), Obj_2.eGet(r_2), fIds, roles) \wedge \quad (9)$$
$$\forall a_2 \in Obj_2.getAllAttributes() \mid Obj_2.eGet(a_2).isDefined() \bullet \quad (10)$$
$$\exists a_1 :\in Obj_1.getAllAttributes() | \quad (11)$$
$$(a_2.name = a_1.name \wedge Obj_1.eGet(a_1) = Obj_2.eGet(a_2)) \vee \quad (12)$$
$$(\exists ro : roles | ro.player = Obj_2 \wedge a_2.name \in fIds) \quad (13)$$

We also included two auxiliary methods that take into account the list of features used as identifiers ($fIds$) in the model type (as we said before, this list changes from a domain to another) and a list of variables ($roles$). The method $Booleancontains(M_1, M_2, fIds, roles)$ is used to indicate that $M_2$ is a snippet of $M_1$ considering a set of feature identifiers and pattern variables. With the help of a method with similar signature for objects, it checks if every objects in $M_2$ has a counterpart in $M_1$, checking every association (lines 7-9) and comparing every attribute (lines 10-13). While the attributes are checked, feature identifiers of a pattern variable are ignored (line 13).

However, pattern-matching is not solely about detecting that a pattern snippet exists in a model. We need to cope with the possible bindings of the pattern variables with the *intended model*. For our purposes, we define pattern-matching

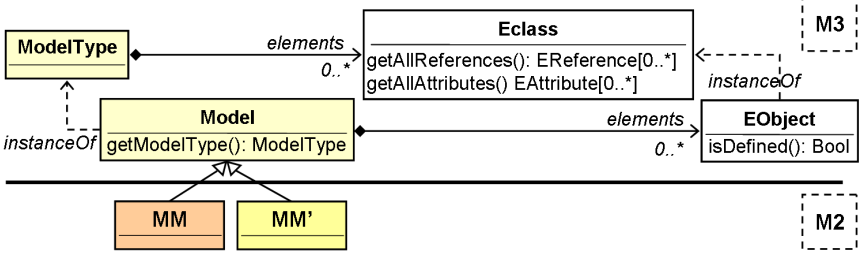


**Fig. 7.** Abstraction level of $MM$ and $MM'$

as the set of possible bindings (tuples with variable name and object) from the pattern variables into the objects of the *intended model*.

$$patternMatching(pat : Pattern, m : Model) : Role[0..*][0..*] = \quad (14)$$
$$\textbf{lets } patsnpt = pat.structure.snippet \wedge vars = pat.structure.variables \wedge \quad (15)$$
$$ids = pat.featureIDName \quad (16)$$
$$\forall snpt : Model \mid contains(m, snpt) \wedge contains(snpt, patsnpt, ids, vars) \wedge \quad (17)$$
$$\nexists minorsnpt : Model \mid minorsnpt! = snpt \wedge contains(snpt, minorsnpt) \wedge \quad (18)$$
$$contains(minorsnpt, patsnpt, ids, vars) \bullet \quad (19)$$
$$\exists setroles \in \textbf{result} \mid \quad (20)$$
$$\forall e_s \in snpt.elements, ro \in vars \mid contains(e_s, ro.player, ids, vars) \wedge \quad (21)$$
$$(\forall (ro.name, e) \in setroles \mid e = e_s) \bullet (ro.name, e_s) \in setroles \quad (22)$$

It the above expression, the *patternMatching* method is formed by two steps, first, in the lines 17-19, the smallest snippets of the *intended model* that contains the pattern structure are found. Then, in lines 20-22, the bindings with the objects of these snippets are finally defined. This is a very general and inefficient definition. We use it here for explanation purposes. The way that we address its implementation is described in the next section.

### 2.4 Model-Snippets During Model Evolution

During a software evolution, an important question that might raises is:

- How useful is a model-snippet (or pattern) when definitions in the meta-model are changed ?

The need for identifying relationships between models, including model-snippets, suggests that we might examine *model-typing* [6]. Indeed, we need to identify a type for the model-snippet in order to verify if a pattern can still be applied in a certain domain, after changes in the meta-model.

For this purpose, we can think in *patternMatching*() (see Fig. 3) as a parametrized type operation, that can be performed to any model that conforms to a criterion. A good starting point for this criterion is to use type checking based on the type relationship *matching* [6] ($<\#$), where any operation $Op[X <\# MM_T]$ parametrized with the group of types $X$ can be successfully performed when $X$ matches to a group of types $MM_T$. Generally speaking, matching[2] in this context is satisfied when all class $c_t$ in $MM_T$ has a respective class $c_x$ in $X$ that matches to it. A class $c_x$ matches to another $c_t$ if: 1) $c_x$ includes all features of $c_t$, 2) and, for all associations of $c_t$ with other class $c_t^{'}$ there is an association in $c_x$ to a class $c_x^{'}$, which contains $c_t^{'}$.

Despite we have presented a pattern-framework that is independent of any meta-model, we can use model type as a pre-step criterion for verifying the validity of a pattern in a specific domain, comparing the model types of the

[2] Do not mix *pattern-matching* with the type relationship called *matching*. We use only *matching* when we refer to the latter in this paper.

pattern *snippet* and the intended model. This criterion can be expressed as a pre-condition of the pattern-matching operation, as follows.

$$patternMatching(pat : Pattern, m : Model) : Role[0..*][0..*]$$
$$\textbf{pre} : pat.structure.snippet.getModelType() < \#m.getModelType()$$

From the expression above, we identify that we can use any meta-model $MM'$ to describe a pattern snippet, provided that $MM' < \#MM$. So, in other words, a model-snippets is still useful if the changes applied to a meta-model $MM$ does not make it less abstract than the $MM'$.

## 3 Implementation

From Sect. 2, we can see how pattern-matching is performed. However, an efficient implementation might not be so easy to construct. Fortunately, this is an extensive topic of research, which has produced several existing languages and APIs with embedded pattern-matching mechanisms [2,7]. For that reason, we have decided to rely on these existing tools as much as possible in order to integrate our ideas and to contribute with existing tools in this research topic.

Our implementation relies on the Kermeta language [8], an executable and object-oriented DSL (Domain Specific Language) for meta-model engineering. Kermeta is built as a conservative extension of EMOF, giving special attention to the specification of abstract syntax, static semantic (OCL) and operational semantics as well with connexion to the concrete syntax. Consequently, an EMF model is seen as a Kermeta model without operational semantics. Through our implementation, we contribute with pattern-matching mechanisms to the meta-model engineering environment available with Kermeta, which includes model transformations, aspect weaving and loading of EMF models.

For our purpose, we have implemented a pattern-matching front-end in Kermeta. This front-end behaves as an abstract interface between our framework for pattern-matching and existing engines with embedded pattern-matching mechanisms. In order to delegate computation to these engines, we require the implementation of a specialised back-end for each engine.

As a proof of concept, we have constructed a back-end that uses a Prolog engine to perform pattern-matching. Using this approach, facts are derived from

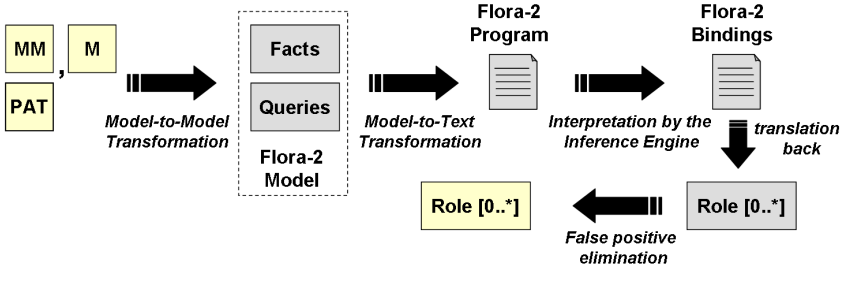


**Fig. 8.** The workflow of pattern-matching implementation using flora2

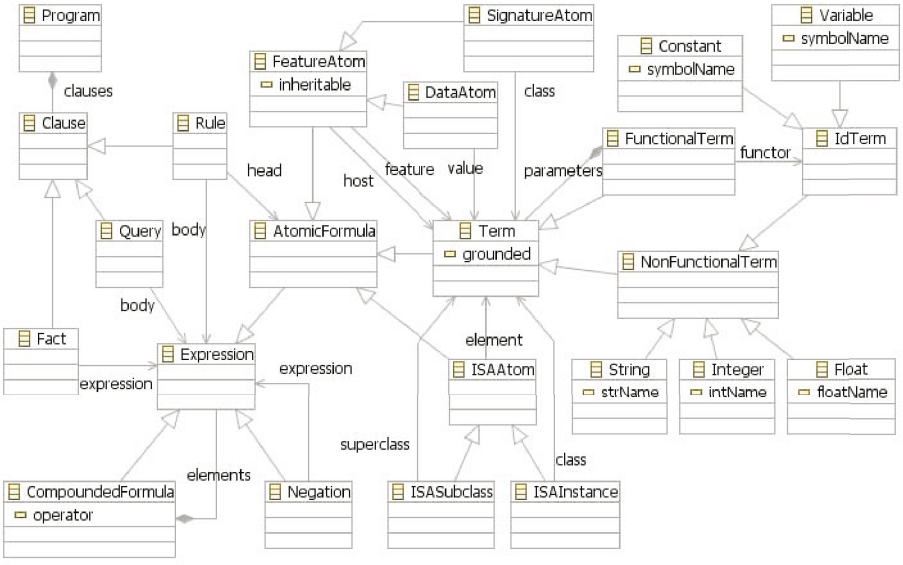


**Fig. 9.** A simplified meta-model of Flora-2

the model that we intent to match and inserted in a knowledge base of the engine. Then, queries are generated to search for a subset of the facts that matches with the pattern. An abstract workflow of this back-end is presented in Fig. 8.

Initially, the front-end inputs the pattern and the *intended model* into the back-end in Kermeta. The later transforms both into an intermediate model in Flora-2, an object-oriented Prolog dialect that is suitable to represent concepts in EMOF [9]; a simplified meta-model for Flora-2 is presented next. In order to interpret this model in Flora-2, we serialise it in the concrete textual syntax of Flora-2. Then, our back-end implementation in Kermenta sends the Flora-2 program to the Prolog Engine trough a Java proxy layer; Kermeta supports a seamless integration with Java programs. The Engine interprets the program and returns the result for our query. From the analysis of this result, we obtain a set of bindings from pattern variables into elements in the *intended model*.

Being object-oriented, Flora-2 offers all necessary counterparts for concepts in EMOF. However, as it is a conservative extension of Prolog, Flora-2 still is a fact-based language. All information about a class or instance is distributed in several atoms[1] (*AtomicFormula*), as it is presented in Fig. 9. Fig. 9 shows a simplified and incomplete meta-model for Flora-2, which only addresses concepts that Flora-2 takes from F-Logic.

Any concept in Flora-2 starts being represented by the most basic atom, a *Term*, which might be classified as a primitive type, a constant or a variant;

[1] Despite *Molecule* constructions permit the direct representation of object and classes and their features, it still is a syntax sugar for a set of related atom in the Knowledge base. Furthermore, it is not represented in our meta-model.

the last one is usually used in queries. Initially any new term might represent a class or an object in the knowledge base. In a class declaration, its super-types are declared through $ISASubClass$ atoms, and features of the class are declared through $SignatureAtom$ atoms. For example, the class $State$ in Fig. 1 is represented by a term with the same name, its superclass by a term $Vertex$, and it is the *host* of a feature *entry* of class *Activity*. These atoms are textually expressed in Flora-2 as $State :: Vertex$ and $State[entry\ *\!\!=\!\!>\ Activity]$; they could also be condensed in a unique molecule $State :: Vertex[entry\ *\!\!=\!\!>\ Activity]$. The type of an object instance is represented through an $ISAInstance$ atom, and the value of its features through $DataAtom$ atoms. So, a state $S1$ with entry $entryAct$ and *exit* $exitAct$, is textually expressed as $S1 :: State$, $S1[entry\ ->\ entryAct]$, $S1[entry\ ->\ entryAct]$.

After transforming the *intended model* and its meta-model into a Flora-2 model, using the meta-model in Fig. 9, the back-end generates a program in Flora-2 in its textual concrete syntax. So, for the example presented in Sect. 1.1, we obtain the following program. For brevity, we do not present all program.

```
// Facts about the meta-model
PseudoState :: Vertex,  State :: Vertex.
FinalState :: State, InitialState :: PseudoState.
...
//Facts about the model
SM1:StateMachine [name -> "sm1", topRegion -> Reg1].
Reg1 [transition => {T1, T2, T3, T4, T5}, vertex => {S0, S1, S2, S3}].
T1:Transition[name -> "T1", source -> S0, target -> S1, trigger -> event1].
T2:Transition[name -> "T2", source -> S1, target -> S1, trigger -> event2].
T3:Transition[name -> "T3", source -> S1, target -> S3, trigger -> event3].
T4:Transition[name -> "T4", source -> S2, target -> S1, trigger -> event4].
T5:Transition[name -> "T5", source -> S1, target -> S2, trigger -> event5].
S0:InitialState[name -> "S0", owner -> Reg1].
S1:State[name -> "S1", entry -> Activity1].
S2:State[name -> "S2", entry -> Activity2].
S3:FinalState[name -> "S3"owner -> Reg1].
```

In the program above, we summarised some facts about subtyping in the meta-model, which we use next for pattern matching. Subtyping is expressed in the form $Class :: SuperClass$. Classes are themselves expressed using atomic clauses (*atoms*) or *molecules*. And, objects are represented in the form $ObjectID : Class[feature_1 -> value, ..., feature_n\ =\!\!>\ setOfValues]$, where the operators $->$ and $=\!\!>$ map a feature into a single and multiple values, respectively.

In this way, if we wish to detect all transitions from two states, excepting self-transitions, we would use the pattern presented in Fig. 10. The transition, its source and target states are presented in the left-hand side of the figure, while the false-positive pattern with a cyclic transition is shown in the right-hand side.

The Flora-2 textual representation of this pattern, considering the false-positive pattern in the right-hand side of figure, is expressed bellow. The query is composed of clauses, conjuncted by commas, with Flora-2 variables (symbols

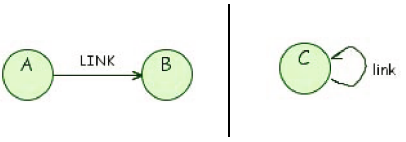


**Fig. 10.** Patterns of state machines

with a ? suffix) to represent all pattern variables. The false-positive is represented with a negation clause $not((...))$.

```
?link : Transition[source -> ?SourceState, target -> ?TargetState],
not ((?link : Transition [source -> ?SameState, target -> ?SameState])),
?SourceState : State, ?TargetState : State, ?SameState : State.
```

The query result is presented as follows. Note that the transition from $S1$ to the final state ($S3$) is also returned, since the type $FinalState$ inherits from $State$ in the state-machine meta-model (see Fig. 1).

| | | |
|---|---|---|
| ?source = S1 | ?source = S2 | ?source = S1 |
| ?target = S2 | ?target = S1 | ?target = S3 |
| ?link = T5 | ?link = T4 | ?link = T3 |

After analysing the set of bindings from the inference engine, which maps variables into elements of the *intended model*, we check constraints over the pattern considering each new set of binding for roles in the pattern. This final checking is useful to eliminate false positives in the pattern. Note that part of these constraints might be anticipated through the representation of patterns for false positives, using the $falsepositive$ feature of the pattern (see Fig. 3). Elements in $falsepositive$ can also be conceptually seen as an negative expression in the query in Prolog, restricting the possible results in the search. Despite the possible use of negative patterns, constraints are always checked, as a post-procedure to filter false positives from the results of the pattern-matching.

After the elimination of false positives, a final set of bindings are presented to the user, or program which invoked the pattern-matching, who can chose among the bindings the most convenient one.

## 4 Related Work

Pattern-matching has been analysed theoretically in various contexts. For meta-model engineering, the most successful results have been achieved in approaches that use graph transformation, such as GReAT[1] and AGG[1]. They rely on graph theories to perform searches in an intermediate graph model that represents the source model. Patterns are also constructed as graphs, which are used to match with model graphs. Differently, but still using an intermediate language, MOLA has an efficient approach for pattern matching [7] through SQL queries in a model repository located in a database.

The unique exception, known by the authors, that proposes a concrete syntax for pattern for graph transformations is reported in [10]. Similar to our approach, this work discuss how a metamodel for patterns can be generated conforming to the meta-model of the model that we wish to match, and how these patterns can visualised using a concrete syntax. Besides its originality, this work does not address negative application conditions and constraints (like in OCL) as we do.

In ATL [2], pattern-matching is used to identify a source element for declarative transformations. Opposite to real pattern-matching, so called pattern-matching in ATL is used in a simplistic way to identify a unique element, rather that a *snippet* of the model. Features related to pattern-matching in all these languages are encompassed in the approach presented in Sect. 2, which solves a common deficiency in non-graph-like transformation languages, as Kermeta.

Concerning the implementation in Sect. 3, pattern-matching implemented via Prolog have been already studied in [3]. However, contrary to our approach, this work has focused in the detection of design patterns in object-oriented programs. While the process steps are very similar, our implementation concerns the application of Prolog in pattern-matching at the model level. The derivated facts from the meta-models and models in this section are also aligned to the formal mapping of class and object into Flora-2 in [9] and other transformations using F-Logic [11]. However, we take as input EMOF classes and objects.

## 5 Conclusion

Nearly all of graph model transformation languages use pattern-matching as the main functional element for defining how the source model components must be detected in model transformations or weavings. For this reason, this work brings an important contribution to Kermeta, permitting the use of pattern-matching mechanisms in its environment. It also brings an alternative between so many existing pattern-matchings using graph-transformations.

In Sect. 2, we also present a simple meta-model framework for patterns, which is beyond any language or implementation. We have shown that this framework might be used in conjunct with any meta-models in order to represent concepts in existing or new domains. We have also discussed how pattern *snippets* might be validated according to the model type of models in these domains. This have shown that indeed, although its limitations, we can use these meta-models to guide the specification of pattern snippets; this also brings some advantages in the use of existing model editors to express model-snippets. This consideration of model type in pattern-matching, and the relation between patterns and the models that they intent to match, have been neglected so far.

An efficient implementation of these ideas is proposed in Sect. 3, through the integration of Kemeta and Flora-2. This is by itself an original contribution, concerning model transformations from arbitrary EMOF model to Flora-2 and the presentation of a, despite simplified, meta-model for Flora-2.

[1] see tfs.cs.tu-berlin.de/agg and repo.isis.vanderbilt.edu/tools

As future work, we plan to match semantic equivalent models. Examples of its need are mainly found in the pattern-matching of behavioural models, since too much semantic information is required. Another future work is to take advantage of transaction logic to directly introduce OCL constraints as Flora-2 facts, in order to improve the overall performance of our implementation. Indeed, such optimisation contributes to a better integration with the Prolog engine, which uses execution strategies with *backtracking* to reduce the exhaustive search.

## References


1. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
2. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop at MoDELS/UML (2005)
3. Prechelt, L., Krämer, C.: Functionality versus practicality: Employing existing tools for recovering structural design patterns. Journal of Universal Computer Science (J.UCS) 4, 866–882 (1998)
4. Clarke, S., Walker, R.J.: Composition patterns: an approach to designing reusable aspects. In: ICSE'01. 23rd International Conference on Software Engineering, Washington, DC, USA, pp. 5–14. IEEE Computer Society, Los Alamitos (2001)
5. Barais, O., Duchien, L., Meur, A.F.L.: A framework to specify incremental software architecture transformations. In: 31st EUROMICRO Conf. on Software Engineering and Advanced Applications, IEEE Computer Society, Los Alamitos (2005)
6. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) (to appear, 2007)
7. Kalnins, A., Celms, E., Sostaks, A.: Simple and efficient implementation of pattern matching in mola tool. In: Baltic DB&IS2006, Vilnius, Lithuania, pp. 159–167 (2006)
8. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
9. Ramalho, F., Robin, J.: Mapping uml class diagrams to object-oriented logic programs for formal. In: 3rd UML Workshop in Software Model Engineering (WiSME 2004) at MODELS/UML'2004, Lisbon, Portugal, pp. 11–15 (2004)
10. Baar, T., Whittle, J.: On the usage of concrete syntax in model transformation rules. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, Springer, Heidelberg (2007)
11. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of mda. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 90–105. Springer, Heidelberg (2002)