Adaptive Transformation Pattern for Architectural Models

Diego Rodríguez-Gracia[1], Javier Criado[1],
Luis Iribarne[1], Nicolás Padilla[1], and Cristina Vicente-Chicote[2]

[1] Applied Computing Group, University of Almería, Spain
{diegorg,javi.criado,luis.iribarne,npadilla}@ual.es
[2] Department of Information Communication Technologies
Technical University of Cartagena, Spain
cristina.vicente@upct.es

Abstract. Model-Driven Engineering (MDE) usually concerns with the production of non-executable models. These models are usually manipulated at design-time by means of fixed model transformations. However, in some situations, models need to be transformed at runtime. Furthermore, the transformations that manipulate these models may also need to change dynamically according to the current execution context and requirements. In this vein, this paper presents a transformation pattern aimed to adapt architectural models at runtime. The transformations that produce this model adaptation are not fixed, but dynamically composed at runtime by selecting the most appropriate set of rules from those available in a repository.

Keywords: Adaptive Transformation, Rule Selection, MDE

1 Introduction

Model Driven Engineering(MDE) is based on the construction of models using formal modeling languages, which can be either general-purpose (e.g., UML) or domain-specific. Models are usually static artifacts. In order to allow models to dynamically evolve, we need to use model transformations. This mechanism enables automatic model redesign and improves model maintainability. Model transformations usually show a static behavior. Such a static behavior prevents models to adapt to requirements not taken into account *a priori*. Therefore, it is necessary to provide model transformations with a dynamic behavior that allows them to vary in time according to new application or user requirements.

The proposal presented in this paper aims to provide model transformations with such a dynamic behavior. In particular, our proposal addresses the adaptation of architectural models by means of transformations that are themselves adapted at runtime [5]. Our architectural model definition is described in a previous work [7]. We present a transformation pattern according to which the transformations that carry out the adaptation are not prepared *a priori*, but dynamically composed at runtime from a rule model. At each transformation

step, this rule model evolves by applying a rule selection algorithm. This algorithm selects the most appropriate set of rules (from those available in a rule repository) according to the current situation. It is worth noting that the goals of this research emerge from the previous results obtained in [7], [8], and [14].

In order to achieve these goals, we will use both *Model-to-Model* (M2M) [1] and *Model-to-Text* (M2T) [2] transformations. Whenever an adaptation of the architectural model is required (e.g., when the user or the system trigger an event), a M2T transformation is invoked. This M2T transformation is an static artifact (i.e., it does not change in time). It takes the current architectural model (containing information about the current context) and generates a M2M transformation, specifically composed to carry out the architectural model adaptation. The input and the output models of this M2M transformation can conform to either the same or different metamodels [9]. The generated M2M transformations contain a set of *transformation rules*. This transformation rules are divided into a *Left-Hand-Side* (LHS) and a *Right-Hand-Side* (RHS). The LHS and RHS refer to elements in the source and the target models, respectively. Both LHS and RHS can be represented through variables, patterns, and logic [9].

We have implemented our M2T transformation using JET [3], while the generated M2M transformations are defined in ATL [15]. We selected ATL as it enables the adoption of an hybrid (of declarative and imperative) M2M transformation approach [9]. In fact, in ATL it is possible to define *hybrid transformation rules* in which both the source and the target declarative patterns can be complemented with an imperative block. It is in this imperative logic where the rule selection algorithm has been implemented. We have also defined a rule metamodel, aimed to help designers: (1) to define correct transformation rules (the metamodel establishes the structure of these rules and how they can be combined), and (2) to store these rules in a repository.

The rest of the article is organized as follows: Section 2 reviews related work. Section 3 details the proposed approach to achieve model transformation adaptation at runtime. In this section, we first describe the transformation pattern that enables to model the structure and composition of the generic elements included in our transformation schema; secondly, we present this transformation schema and describe the processes involved in it; thirdly, we show the transformation rules stored in the repository and focus on the rule selection algorithm. We describe the rule selection process in detail and show the result of appying it in an example repository. To conclude, we describe the process designed to generate the transformations from the selected rule models. Finally, Section 4 outlines the conclusions and future work.

2 Related Work

Nowadays there are different proposals to achieve adaptive transformations for architectural models at runtime. To this end, in [11] the authors developed metatransformations as transformations which operate over other ones, being their source or target, that is to say, they are transformations that produce trans-

formations. However, unlike the proposal developed in [11], our approach has the special feature that new transformations are created to get adaptability in the architectural models (horizontal transformations) rather than make the transformation from PIM to PSM models (vertical transformations). In [10], the architectural models must contain variation and selection criteria so the middleware can automate the transformation. In contrast, we propose to store the adaptation logic in a repository of transformation rules.

Other approaches face the problem of achieving model-adaptability at runtime through high level language implementations. For instance, in [18] the authors used *MAte* and *MUTate* modules implemented as Java programs that are executed inside an OSGi [4] platform. In our case, we achieved the runtime model adaptation and update through model transformations (M2M and M2T). Such transformations are made by means of rules implemented in the ATL model transformation language. One of the features of ATL which made us use this language to implement rules, is that it enables to use explicit rule calls internally as a mechanism for rule integration [16]; thus, rules are assembled so that one rule calls another one.

Different proposals of internal composition techniques for model transformation languages haven been developed. In [21] the authors present an internal composition mechanism of model transformation, implemented in a rule-based model transformation language which uses ATL language as an example. This mechanism enables to make two or more transformations in just one, splitting up a model transformation into multiple transformations. To this end, the authors suggest creating transformation modules that can be either called from other transformation modules or imported from an ATL transformation file. To our opinion, as ATL is the metamodel of these modules, it would be harder to manage and interpret them automatically than use the models and rule repository of our proposal. Thus, we chose to create ATL rules defined by a DSL and dynamically build ATL transformation modules.

On the other hand, in [19] the authors suggested the use of model-to-model transformations to generate as output transformation models in order to adapt or modify an M2M transformation process. Such models can be later turned into ATL transformation files that behave in turn as new transformation modules adapted to the system's requirements. This composition method for transformation process is quite interesting and guarantees well-built transformation modules, since we used the ATL metamodel as reference to generate transformation models; however, these *Higher-Order Transformations* (HOT) [19] are very complex to be built when there are significant rule modifications or when we wish to create an ATL transformation model from a rule model of our system.

In [13] the authors use the term "live transformations" to describe those transformations that increase or decrease (their rules or facts) to move forward in their transformation logic tree; that is, a dependence tree that increases or decreases according to the source model behavior (*i.e.*, a new tree is not built). Declarative rules are built (increased or decreased) from a rule repository to allow changes in runtime transformations subject.

The approach developed in [17] has many aspects in common with ours. It proposes to describe and execute model refactorings based on transformation rules or checked actions where rules have formal parameters that are matched with a model subset. The main difference with our proposal is that we used specific MDD tools, Ecore models [12] instead of UML [6] ones and ATL [15] language rather than Python [20]. We carried out the selection of transformation rules through model transformations.

3 Adaptive Model Transformation

As previously advanced, models created at design time from model definition language are, in principle, static elements. Here we will define design-time architectural models and we want them to be changing and adapting to the system's requirements. Once a specific model has been built (conforming to a metamodel), we can either directly modify its content or use *Model Driven Engineering* (MDE) techniques. In order to modify our architectural models, we based on the MDE methodology so that we can achieve their change and adaptation by using model-to-model (M2M) transformations.

In an M2M transformation, an output model (conforming to a metamodel) is generated from an input model that conforms to a metamodel which may or may not be the same as the output metamodel. Here, we will design an M2M transformation where both the input and output metamodels are the same, the architectural metamodel (AMM). Therefore, this process will turn an architectural model AM_a into another AM_b (Figure 1).

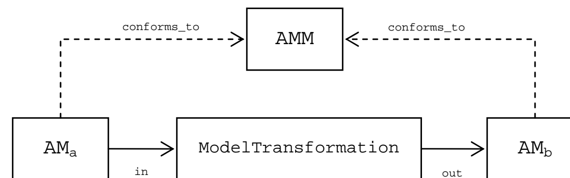

Fig. 1. Architectural Model Transformation

This *ModelTransformation* process enables the evolution and adaptation of architectural models. Its behaviour is described by the rules of such transformation. Thus, if our goal is to make the architectural model transformation not be a predefined process but a process adapted to the system's needs and requirements, we must get the transformation rules to change depending on the circumstances. In order to achieve this goal, we based on the following conditions:

(a) Build a rule repository where all rules that may be applied in an architectural model transformation are stored.
(b) Design a rule selection process that takes as input the repository and generates as output a subset of rules.

(c) Ensure that the rule selection process can generate different rule subsets, depending on the circumstances.
(d) Develop a process that takes as input the selected rule subset and generates an architectural model transformation.
(e) Ensure that both the described processes and their elements are within the MDE framework.

According to the proposed conditions for the implementation of our transformation schema, we observed a variety of similarities and analogies between the elements present here. Such similarities have been generalized and expressed in the transformation pattern described in Section 3.1.

3.1 Transformation Pattern

Building a transformation pattern allows us to model the structure and composition of generic elements that may exist in our transformation schema. Such elements provide us with some information about the types of modules that can be included in possible transformation configurations and how they connect with the rest of the schema elements. Furthermore, this pattern offers us the possibility of changing such schema by creating a different model from the metamodel defined in Figure 2, which has been implemented through EMF [12].

A transformation schema (`TransformationSchema`) is made up of three different types of elements: transformations (`Transformation`), models (`Model`) and metamodels (`Metamodel`). `Metamodel` elements describe the model definitions of the transformation schema. `Model` elements identify and define the system models. `Transformation` elements can be classified into two groups: `M2M` and `M2T`. `M2M` transformations represent model-to-model transformation processes; therefore, they will have one or more schema models associated both as input and output through the `source` and `target` references, respectively. On the other

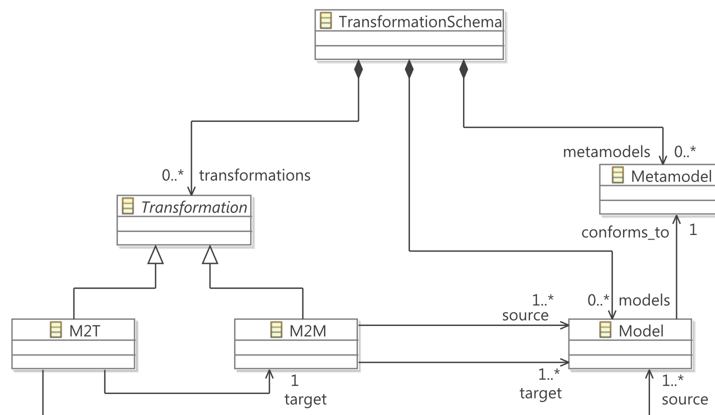

Fig. 2. Transformation Pattern

hand, `M2T` transformations represent the transformation processes that take as input one or more system models (through `source`) and generate as output a model-to-model transformation (through `target`).

3.2 Transformation Schema: An instance of Transformation Pattern

In accordance with the transformation pattern in Section 3.1, we developed our adaptive transformation for architectural models at runtime whose transformation schema is shown in Figure 3. The behaviour and sequence are as follows:

(a) **RuleSelection**, is the rule selection process that starts when an attribute from a defined class in the initial architectural model (AM_i) takes a specific value (*i.e.,* when the user or the system trigger an event). This process, that is carried out at runtime, is obtained as an instance of the `M2M` concept. It takes as input the repository model (RRM) and the AM_i (see step #1 in Figure 3), and generates as output (see step #2 in Figure 3) a rule transformation model (RM_i) for architectural models, being $RM_i \subseteq RRM$.
(b) **RuleTransformation**, is obtained as an instance of the `M2T` concept. It takes as input (see step #3 in Figure 3) the rule model (RM_i) and generates as output (see step #4 in Figure 3) a new transformation process for architectural models at runtime ($ModelTransformation_i$).
(c) **ModelTransformation**, is obtained as an instance of the `M2M` concept and generates as output (see step #6 in Figure 3) a new architectural model at runtime (AM_{i+1}) starting from the initial architectural model (AM_i).

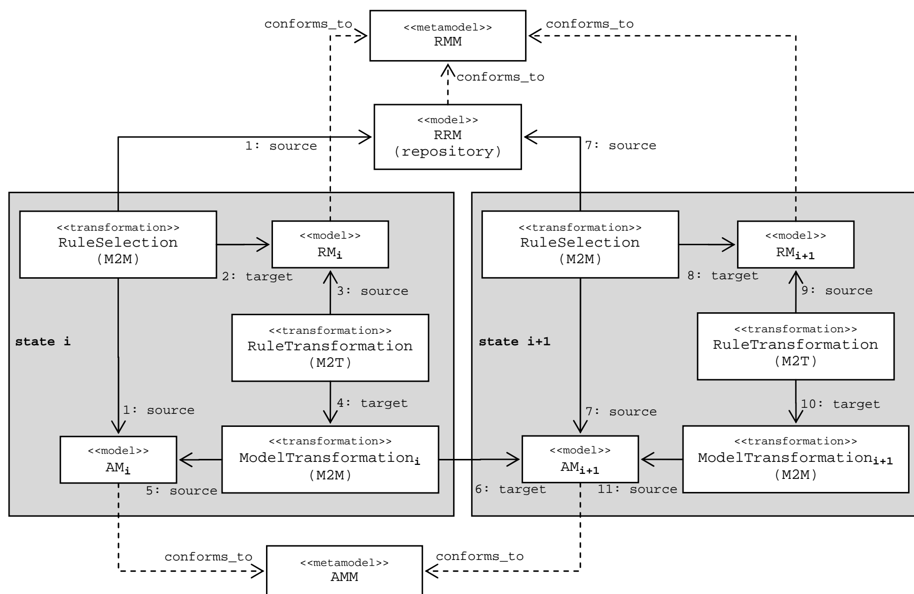

Fig. 3. Transformation Schema

3.3 Transformation Rules: an overview

As previously indicated, our goal is to achieve the adaptability of architectural model transformations at runtime. To this end, and given a transformation rule repository for architectural models (RRM), the system generates transformation rule models (RM_i) that adapt to the properties of the system context at runtime. The transformation rules define the degree of adaptability of our system, as such adaptability depends on the ability of the transformation rule model (RM_i) to modify itself from external events of the system. That is why we focus on the description of the transformation rules and the attributes that affect the rule selection process (*RuleSelection*) and the rule repository (RRM), where the transformation rules of the architectural models are stored.

A. Rule metamodel

Both the transformation rule model (RM_i) and the rule repository (RRM) are defined according to the transformation rule metamodel for architectural model (RMM). In such metamodel, which defines both the transformation rule model (RM_i) and the rule repository (RRM), we will focus on describing the class (`Rule`) which is directly involved with the rule selection logic belonging to the rule model generation process (*RuleSelection*). The class `Rule` (Figure 4) has the following attributes:

— **rule_name**: It is unique and identifies the rule.
— **purpose**: It is defined *a priori* and indicates the purpose of the rule. Only those rules of the rule repository (RRM) whose `purpose` coincides with the `purpose` attribute's value defined in the architectural model (AM_i), will belong to the transformation rule model (RM_i).
— **is_priority**: Boolean. It is established *a priori*. If its value is true in a specific rule of the rule repository (`RRM!Rule.is_priority = true`), it indicates that the rule must always be inserted in the transformation rule model (RM_i), provided that it satisfies the condition detailed in `purpose`.
— **weight**: It is established *a priori*. That rule in the rule repository (RRM) which satisfies the `purpose` condition, has the attribute `is_priority = false` and has the biggest `weight` of all rules satisfying such conditions, will be inserted in the transformation rule model (RM_i).

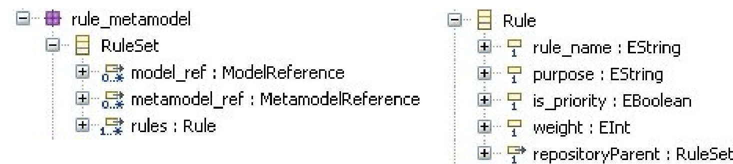

Fig. 4. A piece of rule metamodel

B. Rule repository

The architectural model transformation rules are stored in the rule repository (RRM). It is a model defined according to a rule metamodel (RMM) and is made up of *a priori* transformation rules. As previously mentioned, those rules that fulfil a specific metric are chosen through a rule selection process (*RuleSelection*). Table 1 shows different rules that belong to the rule repository and will be used as an instance in Section 3.4.

Table 1. Example rule repository (RRM)

Rule Repository Model (RRM)			
rule_name	**purpose**	**is_priority**	**weight**
Insert_Component_One	InsertComponent	false	6
Delete_Component	DeleteComponent	true	9
Rename_Component	RenameComponent	false	8
Insert_Component_Two	InsertComponent	false	7
Insert_Component_Three	InsertComponent	true	2

3.4 Rule Selection

After an overview of the transformation rules described in Section 3.3, we studied the transformation process known as *RuleSelection* through which rule models (RM_i) are generated from the rule repository (RRM) to get the transformation adaptation at runtime. According to our transformation schema, this process is obtained as an instance of the `M2M` concept of the transformation pattern (see Section 3.1). Hence, *RuleSelection* is a model-to-model transformation process that takes as input (`source`) the initial architectural model (AM_i) defined in accordance with an architectural metamodel (AMM), and the rule repository model (RRM) defined in compliance with the rule metamodel (RMM). As output (`target`), *RuleSelection* generates the transformation rule model (RM_i) also defined according to the rule metamodel (RMM) (see Figure 5).

The sequence of this M2M transformation process is as follows. The process starts when an attribute of a class defined in the initial architectural model (AM_i) takes a specific value. This class is known as `Launcher`. The selected rule model (RM_i) is generated starting from the rule repository model (RRM). Both models are defined in compliance with the rule metamodel (RMM). This new rule model (RM_i) is made up of a subset of rules existing in the rule repository model(RRM); their `purpose` attribute will coincide with the `purpose` attribute of the class `Launcher`, defined in the initial architectural model (AM_i) and they must fulfil a selection metric based on specific values of the `is_priority` and `weight` attributes. The selection logic is as follows: those rules *a priori* defined as priority (`is_priority = true`) in the rule repository (RRM) will be copied in the transformation rule model (RM_i) regardless of the `weight` value

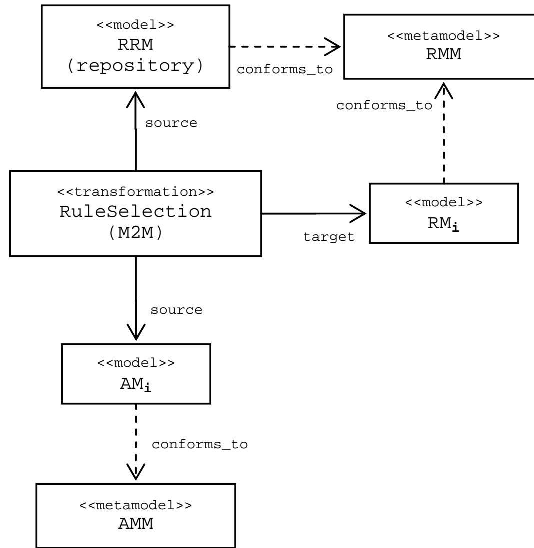

Fig. 5. *RuleSelection* schema

assigned at *state i*, provided that the value of the `purpose` attribute of the rule coincides with the value of the `purpose` attribute of the architectural model (`RRM!Rule.purpose = AMi!Launcher.purpose`). Regarding those rules not defined as priority in the rule repository (`is_priority = false`), the process will copy in the transformation rule model the rule with the biggest `weight` value among all assigned to the rules of the rule repository, where the value of the `purpose` attribute of the rule coincides with the value of the `purpose` attribute of the initial architectural model. This selection logic of the *RuleSelection* process is shown in Table 2.

As a practical example, let's suppose that we take as input the architectural model AM_i where `AMi!Launcher.purpose = 'InsertComponent'`. Let's also suppose that the transformation rule repository model (RRM) is the one specified in Table 1. If, for external reasons, the state of the `running` attribute of the architectural model (AM_i) changed into true (`AMi!Launcher.running = true`) at the *state i*, the *RuleSelection* model-to-model transformation would start. Then, the selected rule model (RM_i) would be generated from the rule repository model (RRM) by selecting the rules with the attribute `purpose = 'InsertComponent'` which have the biggest `weight` or which have their attribute `is_priority = true`, as shown in Table 3.

3.5 Rule Transformation

Starting from the rule model (RM_i) of the selection process described in Section 3.4, the next process involved in the adaptive transformation of our system is known as *RuleTransformation*. Within our transformation schema, this process

Table 2. Selection Logic

```
Input: AMi and RRM
Output: RMi

  If AMi!Launcher.running = true then
    RuleSelection
  End If

RuleSelection
  If RRM!Rule.purpose = AMi!Launcher.purpose then
    If RRM!Rule.is_priority = true then
      RMi!Rule ← RRM!Rule
    Else
      For RRM!Rule.purpose = AMi!Launcher.purpose
        If RRM!Rulen.weight > RRM!Rulen+1.weight
          RMi!Rulen ← RRM!Rulen
        EndIf
      EndFor
    EndIf
  EndIf
```

Table 3. Model of selected rules (RM_i)

Rule Model (RM_i)			
rule_name	**purpose**	**is_priority**	**weight**
Insert_Component_Two	InsertComponent	False	7
Insert_Component_Three	InsertComponent	True	2

is obtained as an instance of the M2T concept of the transformation pattern (see Section 3.1). Therefore, the *RuleTransformation* process is a model-to-text transformation process that takes as input (source) the rule model selected by the *RuleSelection* process and generates as output (target) a model-to-model transformation file (see Figure 6).

The main goal here is to generate a M2M transformation that is responsible for changing the system's architectural models (*ModelTransformation*$_i$). As indicated in our transformation pattern, this new transformation is an instance of the M2M concept that takes as input (source) an architectural model (AM_i) and generates as output (target) another architectural model (AM_{i+1}). Since the rule models of the *RuleSelection* process will be changing depending on the system's requirements, the *RuleTransformation* process (that takes as input these models) is responsible for creating a runtime architectural model transformation that contains new rules considered to be necessary. Hence, this *ModelTransformation*$_i$ process will achieve the adaptation of the architectural models at runtime. As an example, Figure 7 shows a fragment of a rule model generated through the *RuleSelection* process. Here, the information dealing with the input and output models is modeled, as well as the metamodel in which such models are defined.

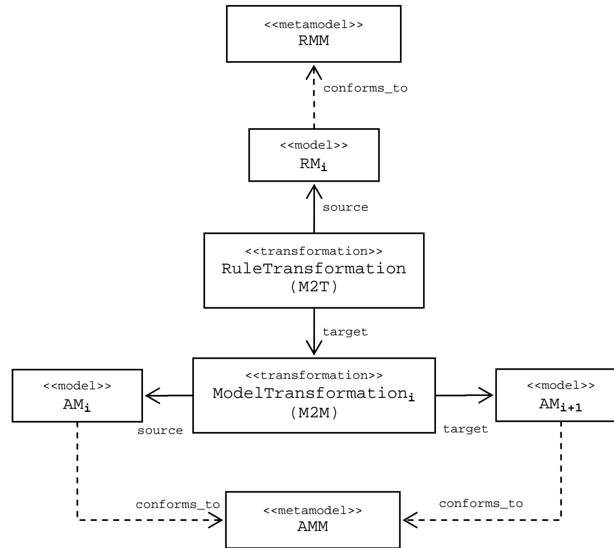

Fig. 6. *RuleTransformation* schema

In Table 4, we can observe the code fragment of the *RuleTransformation* process that is responsible for transforming the part of the model shown in the Figure 7. This part of the model-to-text transformation [2], generates the header section of the ATL transformation file of the $ModelTransformation_i$ process. In every element of the selected rule model (RM_i) there is a part of the M2T transformation of the *RuleTransformation* process that is in charge of translating the rules and any other necessary information into the ATL code, which constitutes the M2M transformation of the $ModelTransformation_i$ process.

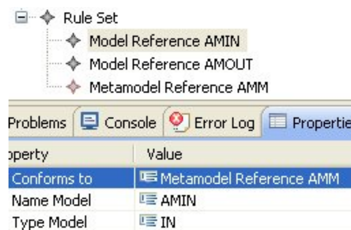

Fig. 7. Example Rule Model extraction

Despite the *RuleTransformation* process has been developed in order to turn rule models into transformation processes applied to architectural models, it is extendable to generate any type of M2M transformation, which is executed on a rule model defined in compliance with the rule metamodel.

Table 4. Transformation example of the *RuleTransformation* process

Portion of transformation M2T

```
module t1;

create
<c:iterate var="model_ref" select="/RuleSet/model_ref[@model_type =
 'OUT']" delimiter=",">
 <c:get select="$model_ref/@model_name"/> :
 <c:get select="$model_ref/conforms_to/@metamodel_name"/>
</c:iterate>
from
<c:iterate var="model_ref" select="/RuleSet/model_ref[@model_type =
 'IN']" delimiter=",">
 <c:get select="$model_ref/@model_name"/> :
 <c:get select="$model_ref/conforms_to/@metamodel_name"/>
</c:iterate>
;
```

M2M generated

```
module t1;

create
  AMOUT : AMM
from
  AMIN : AMM
;
```

4 Conclusions and future work

Here we presented our proposal of adaptive transformations for architectural models at runtime. Our scope are architectural models which represent user interfaces made up of UI components [7]. Thus, we developed a transformation pattern that enables to model the structure and composition of the generic elements that may exist in our transformation schema. With this pattern, it is also possible to change the transformation schema by creating a different model starting from the metamodel that defines it. This provides our proposal with a high degree of flexibility and scalability. We got the transformation rules to change depending on the possible circumstances. Therefore, the transformation rules define the degree of adaptability of our system; such adaptability is determined by the ability of the transformation rule model (RM_i) to modify itself in view of external events of the system, where both the degree and scope of adaptability are also defined by means of the rule selection logic.

As future work, we intend to achieve a higher degree of adaptability for our proposal. To this end, we suggest providing the generation process of transformation rule models with a more adaptive behavior. Thus, we will take into account, in the selection logic, factors that provide new rule selection criteria to

get a higher degree of adaptability in transformations: use frequency of transformation rules, rule weight management policy, etc. We also intend to possibly carry out, through HOT [19], the process by which at runtime we turn rule models into transformation processes applied to architectural models. Once the required adaptability level is reached, and using the scalability degree of our proposal, we'll focus on providing our system with a decision-making technique to be able to manipulate the rule repository so that the system can evolve at runtime and adapt itself to the interaction with the user. Moreover, another improvement we wish to include in our system, is the development of an editing tool for transformation rules. Thus, we would be able to manage both the rule repository and each rule model generated through a friendly graphical interface in a similar way to that in [18]. On the other hand, this tool would allow us to execute the rule selection process to check which rules are selected from the repository and the context information.

Acknowledgments. This work has been supported by the EU (FEDER) and the Spanish Ministry MICINN under grant of the TIN2010-15588 and TRA2009-0309 projects, and also by the JUNTA ANDALUCÍA (proyecto de excelencia) ref. TIC-6114, `http://www.ual.es/acg`.

References

1. Eclipse Modeling Project – Model to Model Transformations. `http://www.eclipse.org/modeling/m2m/`.
2. Eclipse Modeling Project – Model to Text Transformations. `http://www.eclipse.org/modeling/m2t/`.
3. Eclipse Java Emitter Templates (JET). `http://www.eclipse.org/modeling/m2t/?project=jet`.
4. OSGi – The Dynamic Module System for Java. `http://www.osgi.org/`.
5. Blair, G., Bencomo, N., France, R.B.: Models@run.time (Special issue on Models at Run Time). Computer, 40(10):22–27 (2009)
6. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide. Addison-Wesley Professional (2005)
7. Criado, J., Vicente-Chicote, C., Iribarne, L., Padilla, N.: A Model-Driven Approach to Graphical User Interface Runtime Adaptation. Models@Run.Time, CEUR-WS Vol 641 (2010)
8. Criado, J., Padilla, N., Iribarne, L., Asensio, J.: User Interface Composition with COTS-UI and Trading Approaches: Application for Web-Based Environmental Information Systems. Knowledge Management, Information Systems, E-Learning, and Sustainability Research, WSKS'2010, Part I, CCIS 111, pp. 259–266, Springer-Verlag Berlin (2010)
9. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, pp. 1–17. Citeseer (2003)
10. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software, 23(2):62–70 (2006)

11. Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. Computer, 39(2):51–58 (2006)
12. Gronback, R.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional (2009)
13. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. Model Driven Engineering Languages and Systems, pp. 321–335 (2006)
14. Iribarne, L., Padilla, N., Criado, J., Asensio, J., Ayala, R.: A Model Transformation Approach for Automatic Composition of COTS User Interfaces in Web-Based Information Systems. Information Systems Management, 27(3):207–216 (2010)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming, 72(1-2):31–39 (2008)
16. Kurtev, I., van den Berg, K., Jouault, F.: Rule-based modularization in model transformation languages illustrated with ATL. Science of Computer Programming, 68(3):138–154 (2007)
17. Porres, I.: Rule-based update transformations and their application to model refactorings. Software and Systems Modeling, 4(4):368–385 (2005)
18. Serral, E., Valderas, P., Pelechano, V.: Supporting runtime system evolution to adapt to user behaviour. In: Advanced Information Systems Engineering, pp. 378–392. Springer (2010)
19. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Model Driven Architecture-Foundations and Applications, pp. 18–33. Springer (2009)
20. van Rossum, G.: Python language reference manual. Network Theory Ltd. (2003)
21. Wagelaar, D.: Composition techniques for rule-based model transformation languages. Theory and Practice of Model Transformations, pp. 152–167 (2008)