Runtime Adaptation of Architectural Models: An Approach for Adapting User Interfaces

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] Dpt. of Info. Communication Technologies, Tech. University of Cartagena, Spain
cristina.vicente@upct.es

Abstract. Traditional techniques of model-driven development usually concern 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. Moreover, the transformations handling these models could be provided with a dynamic behavior enabling the adaptation to the current execution context and requirements. In this vein, this paper defines a transformation pattern designed for flexible model transformation that can be dynamically composed by selecting the appropriate transformation rules from a rule repository, which is also represented by a model. The rules in the repository are updated at each step of adaptation to improve later rule selection. We chose the domain of user interfaces, specified as component-based architectural models, as our case study.

Keywords: UI, Adaptive Transformation, Rule Selection, MDE.

1 Introduction

In *Model-Driven Engineering* (MDE), transformations to enable model refinement (commonly, into other models or into code) are usually composed and executed at design-time. Furthermore, models are usually static artifacts and model transformations allows us to provide such elements with a dynamic behaviour. Recently, runtime model transformations are being increasingly used as a means of enabling the so-called *executable models* or *models@runtime*. In this context, transformations are used to adapt the models dynamically, although in most cases, they usually show a static behaviour. Such a static behavior prevents models to adapt to requirements not taken into account *a priori*. In order to enable model transformations to evolve at runtime, we need to provide them with a dynamic behaviour.

In this paper we aim to provide model transformations with such a dynamic behavior that allows them to vary in time according to the context. Specifically, our approach addresses the adaptation of architectural models by means of transformations that are themselves adapted at runtime [1]. Our architectural model definition is described in a previous work [2]. The transformations are are dynamically composed *at runtime* by selecting the most appropriate transformation rules from a rule repository according to the current situation. This repository is also updated at each adaptation stage in order to improve the following rule selection process. The transformation pattern proposed has been made as flexible as possible, enabling designers to instantiate it by adding or removing elements, and thus allowing them to customize their runtime model adaptation processes. An example instance of this pattern for dynamically adapting component-based user interface models is described below to illustrate the proposed approach. It is worth noting that the goals of this research emerge from the previous results obtained in [2], [3], and [4].

To achieve our intended goals, we made use of both M2M and M2T transformations [5]. At runtime, when the system detects a need for adaptation (e.g., when the context properties change, or when the user or the system trigger a certain kind of event), an M2M transformation is invoked. This takes the rule repository and the current model (the one to be adapted) as inputs, and selects the most appropriate rules to be executed from the repository according to the context information available in the model. The generated M2M transformations contain a set of *transformation rules*. These 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 [5].

Then, an M2T process generates an M2M transformation out of the rules selected. Our M2T transformation was implemented using JET[6], and generates M2M transformations defined in ATL [7]. We selected ATL because it enables the adoption of an hybrid (of declarative and imperative) M2M transformation approach. 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 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.

We have chosen the domain of user interfaces as part of a project of the Spanish Ministry to develop adaptive user interfaces at runtime, as there is a recognized and increasingly growing need to provide these artifacts with dynamic adaptation capabilities. Here, user interfaces are specified using component-based architectural models (each UI element is represented by a component). These models may vary at runtime due to changes in the context—e.g., user interaction, a temporal event, visual condition, etc. Hence, our proposal is useful to adapt component-based architecture systems at runtime (such as user interfaces based on components) by means of models and traditional techniques of model-driven engineering. Our approach presents two main **advantages** concerning the adaptation of component-based architectural models: (a) the model transformation applied to the architectural model is not fixed, but dynamically composed at runtime, and (b) this composition is made by selecting the most appropriate set of rules (according to the current context) from those available in a repository, making the adaptation logic for the architectural models be upgradable by changing the rule repository, making it possible to change the adaptation logic by adding/removing/changing the rules in the repository.

The rest of the article is organized as follows: Section 2 introduces the goal of adapting component-based user interfaces. Section 3 presents the proposed transformation pattern. In Section 4 we detail the proposed approach to achieve model transformation adaptation at runtime. Section 5 reviews related work. Finally, Section 6 outlines the conclusions and future work.

2 A Running Example: User Interface Adaptation

The main objective of our proposal is to achieve the adaptation of user interfaces at runtime. Specifically, we are interested in studying the evolution of simple and friendly User Interfaces (UI) based on software components, in a similar way iGoogle widget-based user interfaces do (i.e., a set of UI components). In this context, user interfaces are described by means of *architectural models* containing the specification of user-interface components. These architectural models (which represent the user interfaces) can vary at runtime due to certain changes in the context —e.g., user interaction, a temporal event, visual condition, etc.

For instance, let's suppose two users in the system which are performing a communication task by means of a chat, an email, an audio and a video high quality with other users. Consequently, the graphical user interface offered by the system contains the UI components that provided these services. Let's suppose now that a new user profile role is connected to the system, and which requires the use of new services. This requirement involves the user interface automatically change to adapt its architecture to the new situation: i.e., removing the video high quality while a video low quality component, a blackboard component and a filesharing component are inserted (Figure 1).

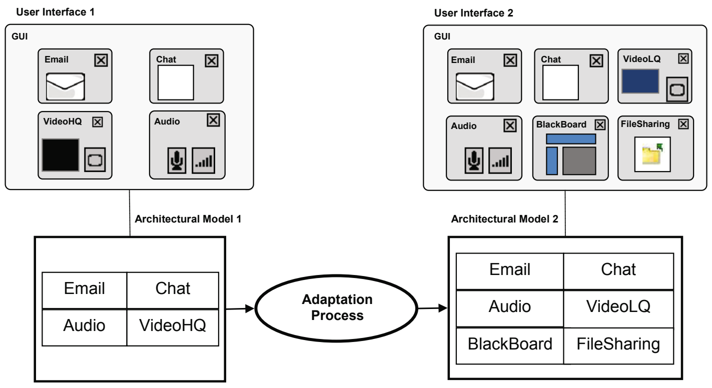


Fig. 1. User Interface Adaptation

Figure 1 illustrates the adaptation process that is performed at the architectural model level representing the user interfaces. Once the new architectural model is obtained (after the adaptation process occurs) a regeneration process is executed to show those software components of the adapted user interface. This regeneration process is not described in this paper, focusing only here on the model transformation process adapting the architectural models, and on how this M2M transformation is dynamically produced from a rule repository (the aims of this paper).

3 Model Transformation Pattern

As previously stated, design-time models are, in principle, static artifacts. Nevertheless, we will define design-time architectural models that will be changed and adapted to the requirements of the system at runtime. In order to modify our architectural models, we follow an MDE methodology so that we can achieve their change and adaptation by M2M transformations. We will design an M2M transformation where both the input and output metamodels are the same: the abstract architectural metamodel (AMM). Therefore, this process will turn an abstract architectural model AM_a into another AM_b (Figure 2).

This *ModelTransformation* process enables the evolution and adaptation of architectural models. Its behaviour is described by the set of rules it contains. 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.

The previous steps involved in the adaptation process share a number of similarities. In order to organize and exploit them, we define a 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

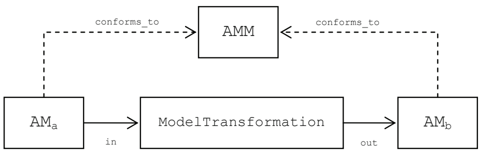


Fig. 2. Architectural Model Transformation

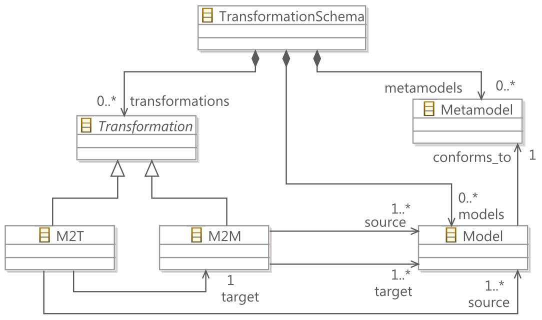


Fig. 3. Transformation Pattern

that can be included in possible transformation configurations and how they connect with the other elements of the schema. Furthermore, this pattern offers us the possibility of changing such schema by creating a different model from the metamodel defined in Figure 3, which has been implemented using EMF [8].

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 hand, `M2T` transformations represent the transformation processes that take one or more system models as their input (through `source`) and generate a textual file that, in our case, corresponds to a M2M transformation (through `target`).

4 Adaptive Model Transformation

The transformation schema enabling the runtime adaptation of architectural model, proposed in this paper, is an instance of the transformation pattern described in Section 3. This schema, illustrated in Figure 4, comprises the following three steps (repeatedly executed at each adaptation step):

(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 4), and generates as output (see step #2 in Figure 4) a rule transformation model (RM_i) for architectural models, being $RM_i \subseteq RRM$.
(b) **RuleSelectionLog (RSL)**, is an instance of the `M2M` concept. Its input is the repository model (RRM) and the selected rule model (RM_i) (step #3),

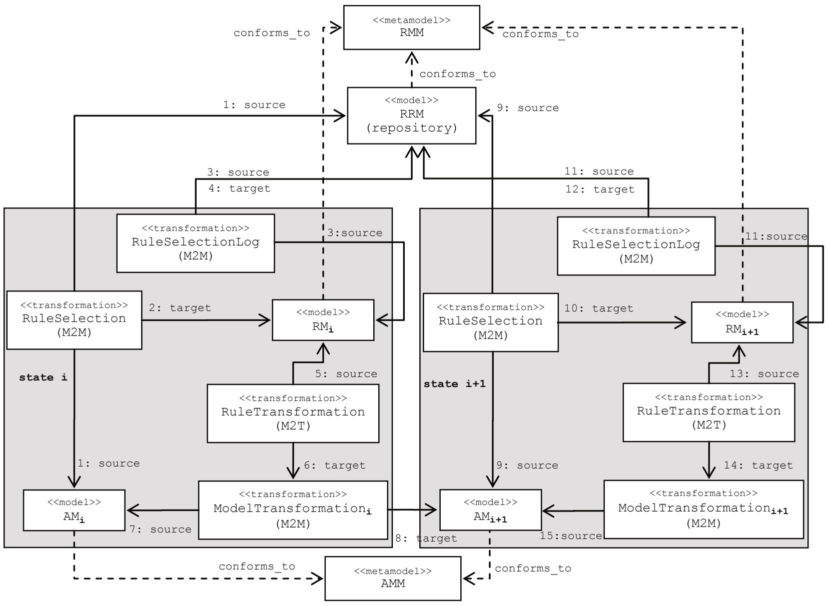


Fig. 4. Transformation Schema

and it generates (step #4) the updated rule repository model for the next adaptation as output.

(c) **RuleTransformation**, is obtained as an instance of the `M2T` concept. It takes as input (step #5) the rule model (RM_i) and generates as output (step #6) a new transformation process for architectural models at runtime ($ModelTransformation_i$).

(d) **ModelTransformation**, is obtained as an instance of the `M2M` concept and generates as output (step #8) a new architectural model at runtime (AM_{i+1}) starting from the initial architectural model (AM_i).

4.1 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.

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 RM_i and RRM transformations, 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` 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 one of the values of the the `purposes` attribute defined in the architectural model (AM_i), will belong to the transformation rule model (RM_i).
— **is_priority**: It is established *a priori.* If its value is `true` in a specific rule of the rule repository, it indicates that the rule must always be inserted in the transformation rule model (RM_i) regardless of its weight, 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).
— **run_counter**: Indicates the number of times the `purpose` of the rule has matched one of the values of the the `purposes` attribute defined in the architectural model (AM_i).
— **selection_counter**: Indicates the number of times the rule has been selected by the *RuleSelection* process to generate the RM_i.
— **ratio**: Indicates the frequency of using a transformation rule (the result of dividing `selection_counter` by `run_counter`).

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 4.2.

4.2 Rule Selection

After an overview of the transformation rules described in Section 4.1, 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). 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

Table 1. Example rule repository (RRM)

rule_name	purpose	is_priority	weight	ratio	run_c	selec_c
Insert_Chat	InsertChat	true	2.0	1.0	3	3
Insert_Audio	InsertAudio	false	4.0	1.0	2	2
Insert_Video1	InsertVideoLowQ	false	11.0	0.5	2	1
Insert_Video2	InsertVideoLowQ	true	6.0	0.5	2	1
Insert_Video3	InsertVideoHighQ	false	9.0	1.0	3	3
Insert_BlackBoard1	InsertBlackBoard	false	6.0	0.5	2	1
Insert_BlackBoard2	InsertBlackBoard	false	4.0	0.5	2	1
Insert_FileSharing	InsertFileSharing	false	3.0	0.0	0	0
Delete_Chat	DeleteChat	true	3.0	0.0	0	0
Delete_Audio	DeleteAudio	true	3.0	0.0	0	0
Delete_Video	DeleteVideo	true	3.0	0.0	0	0
Delete_BlackBoard	DeleteBlackBoard	true	3.0	0.0	0	0
Delete_FileSharing	DeleteFileSharing	true	3.0	0.0	0	0

output (`target`), *RuleSelection* generates the transformation rule model (RM_i) also defined according to the rule metamodel (RMM) (see Figure 4).

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`. Then, the RM_i is generated starting from the 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 one of the `purposes` attribute of the class `Launcher`, defined in the 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 RRM will be copied in the selected rules model (RM_i) regardless of the `weight` value assigned at *state i*, provided that the value of the `purpose` attribute of the rule coincides with one of the values of the `purposes` attribute of the architectural model (`AMi!Launcher.purposes contains RRM!Rule.purpose`). Regarding those rules not defined as priority in the rule repository (`is_priority = false`), the process will copy in the RM_i 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 one of the values of the `purposes` attribute of the initial architectural model. This selection logic of the *RuleSelection* process is shown in Table 2.

As an example, let us suppose the following architectural model AM_i where `AMi!Launcher.purposes = ['DeleteVideo','InsertVideoLowQ', 'Insert-BlackBoard','InsertFileSharing']`. We assume 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* 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 = 'DeleteVideo' or 'InsertVideoLowQ' or 'InsertBlackBoard' or 'InsertFileSharing'`, which have the biggest `weight` or which have their attribute `is_priority = true`, as shown in Table 3.

Table 2. Selection Logic

```
Input: AM_i and RRM Output: RM_i
  if $AM_i$!Launcher.running = true then
     RuleSelection
  end if

RuleSelection
 1: for $n = 1 \rightarrow RRM.size$ do
 2:    if $AM_i$!Launcher.purposes[$EString_1..EString_n$] contains RRM!$Rule_n$.purpose then
 3:       if RRM!$Rule_n$.is_priority = true then
 4:          $RM_i$.add(RRM!$Rule_n$)
 5:       else
 6:          if $\exists!$ j, j $\in$ 1..$RRM.size$/ RRM!$Rule_j$.weight > RRM!$Rule_n$.weight then
 7:             $RM_i$.add(RRM!$Rule_n$)
 8:          end if
 9:       end if
10:    end if
11: end for
```

Table 3. Model of selected rules (RM_i)

rule_name	purpose	is_priority	weight
Insert_Video2	InsertVideoLowQ	true	6.0
Insert_BlackBoard1	InsertBlackBoard	false	6.0
Insert_FileSharing	InsertFileSharing	false	3.0
Delete_Video	DeleteVideo	true	3.0

4.3 Rule Selection Log (RSL)

When the selected rule model (RM_i) has been acquired by *RuleSelection*, the next step is to update the rule repository execution log. This step is necessary to update the attributes representing the frequency with which the rules are used in the various executions of the system, which is given in the weight attribute of the rule that may be used by *RuleSelection* as explained in Section 4.2. Thus, *RuleSelectionLog* (*RSL*) is an M2M transformation process with the selected rule model (RM_i) and the rule repository model (*RRM*) as input (`source`), which generates the updated rule repository model as an output (`target`), as shown in Figure 4.

Moreover, *RSL* transformation uses `bonus` and `penalty` coefficients, which add or subtract weight, respectively, and which are defined *a priori*, applying them depending on whether the rule is selected or not. The *RSL* logic is applied to the rule repository (*RRM*) as shown in Table 4. If the rule is in the selected rule model (RM_i), it is modified in the rule repository model, increasing the process execution counter and the selection counter, and updating the ratio and the weight (lines #3–#6). If the rule is not in the RM_i, but its `action` is the same, it is modified in *RRM*, increasing the execution counter, and updating the ratio and the weight (lines #9–#11).

Following this logic, the *RSL* highlights transformation rule use frequency (`ratio`), so the `weight` of a rule, although defined *a priori*, is determined by the `ratio` with which the rule has been selected by *RuleSelection* to generate the RM_i. Similarly, not using the rule influences its `weight` negatively. Therefore,

Table 4. *RSL* pseudocode

```
process $RSL$
1:  for $n = 1 \to RRM.size$ do
2:    if $RM_i$ contains $RRM[n]$ then
3:      $RRM[n].run\_counter \leftarrow RRM[n].run\_counter + 1$
4:      $RRM[n].selection\_counter \leftarrow RRM[n].selection\_counter + 1$
5:      $RRM[n].ratio \leftarrow RRM[n].selection\_counter / RRM[n].run\_counter$
6:      $RRM[n].weight \leftarrow RRM[n].weight + RRM[n].ratio * RRM.bonus$
7:    else
8:      if $RM_i.action = RRM[n].action$ then
9:        $RRM[n].run\_counter \leftarrow RRM[n].run\_counter + 1$
10:       $RRM[n].ratio \leftarrow RRM[n].selection\_counter / RRM[n].run\_counter$
11:       $RRM[n].weight \leftarrow RRM[n].weight - RRM[n].ratio * RRM.penalty$
12:     end if
13:   end if
14: end for
```

Table 5. RRM after RSL

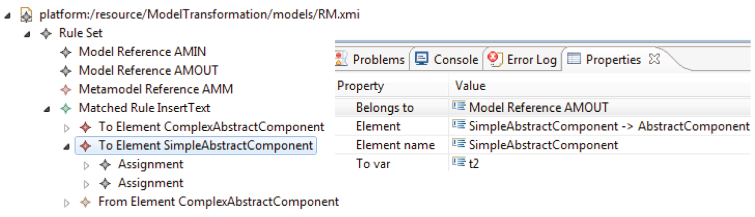
rule_name	purpose	weight	ratio	run_c	selec_c
Insert_Chat	InsertChat	2.0	1.0	3	3
Insert_Audio	InsertAudio	4.0	1.0	2	2
Insert_Video1	InsertVideoLowQ	11.0→10.33	0.5→0.33	2→3	1
Insert_Video2	InsertVideoLowQ	6.0→7.65	0.5→0.66	2→3	1→2
Insert_Video3	InsertVideoHighQ	9.0	1.0	3	3
Insert_BlackBoard1	InsertBlackBoard	6.0→7.65	0.5→0.66	2→3	1→2
Insert_BlackBoard2	InsertBlackBoard	4.0→3.33	0.5→0.33	2→3	1
Insert_FileSharing	InsertFileSharing	3.0→5.5	0→1.0	0→1	0→1
Delete_Chat	DeleteChat	3.0	0.0	0	0
Delete_Audio	DeleteAudio	3.0	0.0	0	0
Delete_Video	DeleteVideo	3.0→5.5	0.0→1.0	0→1	0→1
Delete_BlackBoard	DeleteBlackBoard	3.0	0.0	0	0
Delete_FileSharing	DeleteFileSharing	3.0	0.0	0	0

by this logic, the transformation rules used in *ModelTransformation$_i$* (Figure 4) are adapted to system behavior. Although the attribute values involved in the rule repository model (*RRM*) are set *a priori*, at $state = i$, the values depend on previous system behavior up to that point ($\forall state < i$).

As a practical example, let us assume the rule repository (*RRM*) in Table 1 and the selected rule model (RM_i) generated by *RuleSelection* (Table 3). The output of this transformation would update the rule repository in Table 5.

4.4 Rule Transformation

The last process required for our adaptive transformation when *RuleSelection* and *RSL* have been executed is called *RuleTransformation*. This process is an instance of the `M2T` concept in the transformation pattern (Figure 3), in which the `source` is the selected rule model, and the target is a model-to-model transformation file code (see Figure 4). Thus, the transformation file generated by this process is an instance of the `M2M` concept (Figure 3) having an architectural model (AM_i) as the `source` and generating the new architectural model (AM_{i+1})



```
platform:/resource/ModelTransformation/models/RM.xmi
  Rule Set
    Model Reference AMIN
    Model Reference AMOUT
    Metamodel Reference AMM
    Matched Rule InsertText
      To Element ComplexAbstractComponent
      To Element SimpleAbstractComponent
        Assignment
        Assignment
      From Element ComplexAbstractComponent
```

Property	Value
Belongs to	Model Reference AMOUT
Element	SimpleAbstractComponent -> AbstractComponent
Element name	SimpleAbstractComponent
To var	t2

Fig. 5. Example Rule Model extraction

as output (`target`). Since the rule models in the *RuleSelection* process change depending on the context and system requirements, the *RuleTransformation* process (that takes these models as input) creates runtime architectural model transformations containing the new rules considered to be necessary. Hence, this *ModelTransformation*$_i$ process adapts the architectural models at runtime.

For example, Figure 5 shows a rule model extraction generated by *RuleSelection* in which the information dealing with the input and output models is

Table 6. Example of transformation by 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>;
<c:iterate var="rules" select="/RuleSet/rules">
 <c:if test="$rules[self :: MatchedRule or self :: LazyRule]">
<c:if test="$rules[self :: MatchedRule]">rule</c:if>
<c:if test="$rules[self :: LazyRule]">lazy rule</c:if>
<c:get select="$rules/@rule_name"/>
{
 <c:include template="templates/fromElements.jet" passVariables="rules"/>
 <c:include template="templates/toElements.jet" passVariables="rules"/>
 <c:include template="templates/doBlock.jet" passVariables="rules"/>
}
 </c:if>
</c:iterate>
```

Portion of M2M generated

```
module t1;
create  AMOUT : AMM from  AMIN : AMM;
rule InsertText
{
 from
  f : AMM!ComplexAbstractComponent in AMIN
  ( f.component_name = 'GUI'
  )
 to
  t1 : AMM!ComplexAbstractComponent in AMOUT
  ( component_name <- f.component_name
  ),
  t2 : AMM!SimpleAbstractComponent in AMOUT
  ( component_name <- 'Text',
    component_parent <- t1
  )
}
```

modeled, as well as the metamodel in which such models are defined, and an example rule. Table 6 shows the *RuleTransformation* code fragment responsible for transforming the rule model extraction. This part of the transformation generates the header section of the ATL transformation file and the content of an example rule. For each element of the rule model (RM_i) there is a part of the M2T transformation process that is in charge of translating the rules and any other necessary information into the ATL code, which constitutes the implementation of the M2M transformation of the $ModelTransformation_i$ process.

Even though *RuleTransformation* was developed to convert rule models into transformation processes for architectural models, it is extendable to any type of M2M transformation, which is executed on a rule model defined in compliance with the rule metamodel.

5 Related Work

In recent years, several proposals have attempted to achieve adaptive transformation. For example, in [9], the authors propose an incremental update strategy. These transformations are built dynamically from the rules available in a rule repository. The proposal presented in [10] shares some aspects related to the dynamic selection of transformation rules in common with ours. It proposes model refactoring description and execution based on transformation rules, which have formal parameters matched to a model subset. It differs from our proposal mainly in that we use an M2M transformation to implement this rule selection, not a check algorithm.

In [11], the authors propose a meta-transformation approach, in which transformations are defined that accept other transformations as their input and produce (new or modified) transformations as their output. In our approach, although also a meta-transformation, we use JET to generate transformations providing dynamic model adaptability (horizontal transformation), rather than refine Platform-Independent Models (PIM) into Platform-Specific Models (PSM) (vertical transformation). In [12], the architectural models must contain variation and selection criteria so the middleware can automate the transformation. The authors in [13] define variability models to specify the adaptation logic, separating it from the system functionality. In contrast, we propose to store the adaptation logic in a repository of transformation rules.

Some other approaches, also dealing with runtime software adaptation, rely on the use of high-level programming languages. For instance, in [14], the authors propose a Java-based implementation that is executed within an OSGi [15] platform. The main advantage of using M2M transformations instead of high-level programming languages for dynamic adaptation is that they can evolve dynamically (because they can be treated as models), while programs (whether binary or bytecodes) cannot. In this vein, one of the main benefits of using ATL for our M2M transformations is that it enables the use of explicit rule calls internally as a mechanism for rule integration [16]. This way, rules can be dynamically assembled in such a way that each rule calls up the next.

Finally, other proposals make use of composition techniques for dynamic model transformation. For instance, in [17,18], the authors provide a mechanism (a rule-based model transformation language) for making model transformations out of previously created modules. These transformation modules can either be called up from other modules or imported from an ATL file. On the other hand, in [19,20], the authors suggest using M2M transformations to generate transformation models, which could then be adapted or modified. Such models can later be translated into ATL transformation files that behave, in turn, as new transformation modules adapted to the requirements. But unlike our research work, neither of these approaches makes automatic use of composition techniques to adapt model transformation to changes in context.

6 Conclusions and Future Work

In this paper we have described our approach to provide model transformations with a dynamic behavior allowing them to vary in time according to the new application or user requirements. In particular, our proposal focuses on the adaptation of architectural models at runtime. Our scope are architectural models which represent user interfaces made up of UI components [2]. With this aim, we have developed a transformation pattern for modeling the structure and composition of the transformation schema. The transformation schema can also be changed by creating another model conforming the transformation pattern. This provides our proposal with a high degree of flexibility and scalability. We have also developed an M2M process (*RSL*) that updates the transformation rules stored in the rule repository. This way, along with the rule selection process (*RuleSelection*), the transformation rules are able to change depending on the circumstances. Therefore, the transformation rules stored in the rule repository (*RRM*) define the adaptability of our new system, which depends on the ability of the transformation rule repository to modify itself in view of external system events. The scope of adaptability is 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 in a similar way to that in [14]. 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 under a FPU grant (AP2010-3259), and also by the JUNTA ANDALUCÍA (proyecto de excelencia) ref. TIC-6114, `http://www.ual.es/acg`.

References

1. Blair, G., Bencomo, N., France, R.B.: Models@RT. Computer 40(10), 22–27 (2009)
2. Criado, J., Vicente-Chicote, C., Iribarne, L., Padilla, N.: A Model-Driven Approach to Graphical User Interface RT Adaptation. Models@RT, CEUR-WS 641 (2010)
3. Criado, J., Padilla, N., Iribarne, L., Asensio, J.-A.: User Interface Composition with COTS-UI and Trading Approaches: Application for Web-Based Environmental Information Systems. In: Lytras, M.D., Ordonez De Pablos, P., Ziderman, A., Roulstone, A., Maurer, H., Imber, J.B. (eds.) WSKS 2010. CCIS, vol. 111, pp. 259–266. Springer, Heidelberg (2010)
4. 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)
5. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA Workshop on Generative Tech. in the Context of the MDA, pp. 1–17 (2003)
6. Eclipse Java Emitter Templates (JET), `http://bit.ly/SdxyWw`
7. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
8. Gronback, R.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional (2009)
9. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
10. Porres, I.: Rule-based update transformations and their application to model refactorings. Software and Systems Modeling 4(4), 368–385 (2005)
11. Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. Computer 39(2), 51–58 (2006)
12. 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)
13. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)
14. Serral, E., Valderas, P., Pelechano, V.: Supporting Runtime System Evolution to Adapt to User Behaviour. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 378–392. Springer, Heidelberg (2010)
15. OSGi – The Dynamic Module System for Java, `http://www.osgi.org/`
16. Kurtev, I., van den Berg, K., Jouault, F.: Rule-based modularization in model transformation languages illustrated with ATL. Sci. Comp. Prog. 68(3), 138–154 (2007)
17. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software and Systems Modeling 9(3), 285–309 (2010)

18. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: MDE Languages and Systems, pp. 623–637. Springer (2011)
19. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: MDA-Found. & Applic., pp. 18–33. Springer (2009)
20. Tisi, M., Cabot, J., Jouault, F.: Improving Higher-Order Transformations Support in ATL. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 215–229. Springer, Heidelberg (2010)