# Composing Model Transformations at Runtime
## An Approach for Adapting Component-based 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, Almería, Spain*
[2]*Dpt. of Info. Communication Technologies, Tech. University of Cartagena, Cartagena, Spain*

Keywords:     Adaptive Transformation, Rule Selection, MDE.

Abstract:     Nowadays, large part of the efforts in software development are focused on achieving systems with an as high as possible level of adaptation. With the traditional technique of model-driven development this can be largely accomplished. The inconvenience of these techniques however, is that the models are usually manipulated at design-time by means of fixed transformation. Furthermore, the transformations that manipulate these models cannot change dynamically according to the current execution context. This paper presents a transformation pattern aimed to adapt architectural models at runtime, this means that these models may change dynamically at runtime. The transformations that produce this model adaptation are not fixed, but dynamically composed by selecting the most appropriate set of rules from those available in a repository. As an example scenario for the application of these transformations, we chose architectural models representing component-based UIs.

## 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. 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 which 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 (Blair et al., 2009). The architectural model definition is described in (Criado et al., 2010a). 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 which selects the most

appropriate set of rules (from those available in a rule repository) according to the current situation.

In order to achieve these goals, we use model-to-model and model-to-text transformations[1]. Whenever an adaptation of the architecture is required (e.g., when the user or the system trigger an event), a new adaptation process is invoked. It takes the current architectural model (containing information about the current context) and generates an M2M transformation, specifically composed to carry out the adaptation. We have implemented our M2T transformation using *Java Emitter Templates*[2], while the generated M2M transformations are defined in ATL (Jouault et al., 2008). We selected ATL as it enables the adoption of an hybrid (of declarative and imperative) M2M transformation approach (Czarnecki and Helsen, 2003). 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

---

[1]EMP – http://www.eclipse.org/modeling/
[2]JET – http://www.eclipse.org/modeling/m2t/

structure of these rules and how they can be combined), and (2) to store these rules in a repository.

As a case study, we have chosen the domain of user interfaces as part of a project of the Spanish Ministry to develop adaptive user interfaces at runtime (Criado et al., 2010b). Here, user interfaces are described by means of architectural models that contain the specification of user interfaces components (Iribarne et al., 2010). These architectural models (which represent the user interfaces) can vary at runtime due to changes in the context—e.g., user interaction, a temporal event, visual condition, etc. Therefore, our proposal is useful to adapt component-based architecture systems at runtime (such as user interfaces based on components) by means of models and model-driven engineering techniques. Our approach presents two main advantages concerning the adaptation of 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 appropriate set of rules from those available in a repository, making the adaptation logic for the architectural models be upgradable by changing the rule repository.

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

## 2 UI ADAPTATION

The main objective of our proposal is to achieve the adaptation of user interfaces at runtime. Specifically, we are interested in simple and friendly User Interfaces (UI) based on software components, in a similar way as iGoogle widget-based user interfaces do (i.e., a set of UI components). Thus, user interfaces are described by means of architectural models that contain the specification of user interfaces components. These architectural models (which represent the user



Figure 1: User interface adaptation.

interfaces) can vary at runtime due to changes in the context –e.g., user interaction, a temporal event, visual condition, etc. For example, let us suppose an user that is performing a communication by a chat with other users. Consequently, the graphical user interface offered by the system contains an UI component providing the Chat service. Then, due to causes out of the scope of this work, the system detects the need for adapting the user interface to change the communication method. This adaptation will aim to to remove the chat component while audio and video components will be inserted.

Figure 1 illustrates that the adaptation process is performed at the level of architectural models representing the user interfaces. Once the new architectural model is obtained, it will be executed a regeneration process to show the adapted user interface. This regeneration process is not described because in this paper we focus on the model transformation process adapting the architectural models and how this M2M is dynamically composed from a rule repository.

## 3 ADAPTATION PROCESS

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 by means of automatic changes. 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 $AM_i$ into another $AM_{i+1}$.

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 in-
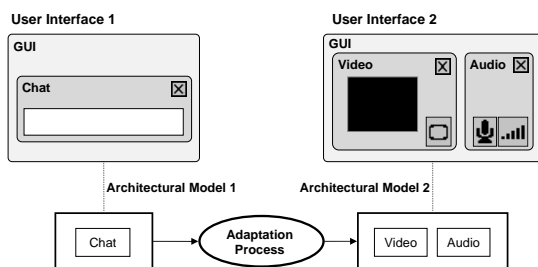
put the selected rule subset and generates an architectural model transformation; and (e) Ensure that both the described processes and their elements are within the MDE framework. According to these steps, we observed a variety of similarities and analogies between the elements present here. Such similarities 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.
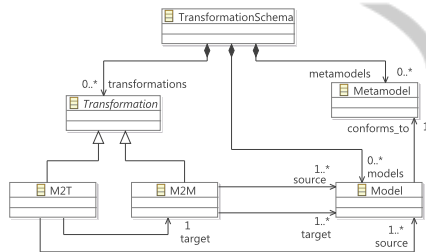


Figure 2: Transformation pattern.

A `TransformationSchema` is made up of three different types of elements: transformations, models and metamodels. `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 as input one or more system models (through `source`) and generate as output a model-to-model transformation (through `target`).

## 3.2 Transformation Schema

In accordance with the transformation pattern in Section 3.1, we developed our adaptive transformation for architectural models at runtime whose transforma-

tion 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 $AM_i$ (step #1 in Figure 3), and generates as output (step #2) a rule transformation model ($RM_i$) for architectural models, being $RM_i \subseteq RRM$.
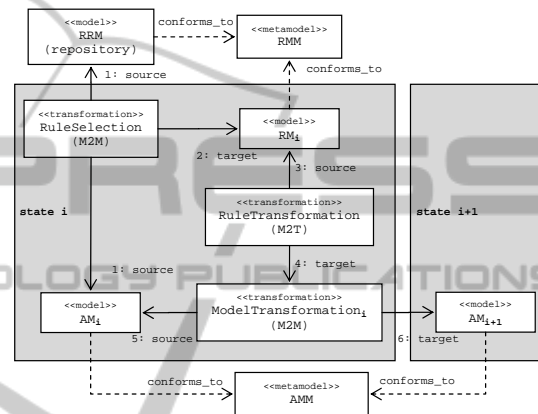


Figure 3: Transformation schema.

(b) **RuleTransformation**, is obtained as an instance of the `M2T` concept. It takes as input (step #3 in Figure 3) the rule model ($RM_i$) and generates as output (step #4) 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 (step #6 in Figure 3) a new architectural model at runtime ($AM_{i+1}$) starting from the initial architectural model ($AM_i$).

## 3.3 Transformation Rules

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 (*RRM*), the system generates the models of selected rules ($RM_i$) adapting the architectural models to the context. That is why we focus on the description of the transformation rules and the attributes that affect the rule selection process (*RuleSelection*) and the *RRM*, where the transformation rules of the architectural models are stored.

Both the $RM_i$ and the *RRM* are defined according to the transformation rule metamodel (*RMM*). In

Table 1: Example rule repository (RRM).

| Rule Repository Model (RRM) | | | |
|---|---|---|---|
| rule_name | purpose | is_priority | weight |
| Insert_Chat | InsertComponentChat | true | 9 |
| Insert_AudioLowQuality | InsertComponentAudioLQ | false | 8 |
| Insert_AudioHighQuality | InsertComponentAudioHQ | true | 5 |
| Insert_VideoLowQuality | InsertComponentVideoLQ | false | 6 |
| Insert_VideoMediumQuality | InsertComponentVideoMQ | false | 7 |
| Insert_VideoHighQuality | InsertComponentVideoHQ | false | 3 |
| Delete_Chat | DeleteComponentChat | true | 1 |
| Delete_Audio | DeleteComponentAudio | true | 1 |
| Delete_Video | DeleteComponentVideo | true | 1 |

Table 2: Model of selected rules (RM$_i$).

| Rule Model (RM$_i$) | | | |
|---|---|---|---|
| rule_name | purpose | is_priority | weight |
| Insert_AudioHighQuality | InsertComponentAudioHQ | true | 5 |
| Insert_VideoMediumQuality | InsertComponentVideoMQ | false | 7 |
| Delete_Chat | DeleteComponentChat | true | 1 |

such metamodel, 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: **(a) rule_name**, which is unique and identifies the rule; **(b) purpose**, which 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$_i$*); **(c) is_priority**, which is boolean typed. 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`; **(d) weight**, which indicates the weight of the rule. 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$_i$*).

The transformation rules that will adapt the architectural models are stored in the rule repository model (*RRM*). It is a model defined according to a rule metamodel (*RMM*) and is made up of the transformation rules. Table 1 shows different rules belonging to the rule repository. As previously mentioned, those rules that fulfil a specific metric are chosen through the rule selection process (*RuleSelection*).

## 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. Hence, *RuleSelection* is an M2M transformation process that takes the initial architectural model (*AM$_i$*) and the rule repository model (*RRM*) as `source`. As `target`, it generates the

transformation rule model (*RM$_i$*).

The sequence of this M2M transformation process is as follows. It starts when an attribute of a class defined in the *AM$_i$* takes a specific value. This class is known as `Launcher`. The *RM$_i$* is generated starting from the *RRM*. This new rule model is made up of a subset of rules existing in the rule repository; 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 *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 transformation rule model the rule with the biggest `weight` value among all assigned to the rules of the *RRM*, where the value of the `purpose` attribute of the rule coincides with one of the values of the `purposes` attribute of the *AM$_i$*.

As an example, let's suppose that we take as input an *AM$_i$* where `AMi!Launcher.purposes = ['DeleteComponentChat','InsertComponentAudioHQ', 'InsertComponentVideoMQ']`. We assume the *RRM* is the one specified in Table 1. If the state of the `running` attribute of the *AM$_i$* changed into `true`, the *RuleSelection* process would start. Then, the *RM$_i$* would be generated by selecting those rules with the `purpose` attribute equals to one of the `purposes` indicated by the `Launcher` which have the biggest `weight` or which have their attribute `is_priority = true`, as shown in Table 2.

## 3.5 Rule Transformation

Starting from the rule model generated by the *RuleSelection*, the next process involved in the adaptive transformation is known as *RuleTransformation*. Within our transformation schema, this process is obtained as an instance of the `M2T` concept of the transformation pattern (Section 3.1). Therefore, the *RuleTransformation* process is an M2T transformation

Table 3: 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;
```

process that takes as input (source) the rule model selected by the *RuleSelection* process and generates as output (target) an M2M transformation file.

The main goal here is to generate an 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 whose source is an architectural model ($AM_i$) and target is 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 them as its input) 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.

Table 3 shows the code fragment of the *RuleTransformation* process transforming the part of the M2T transformation that generates the header section of the ATL file of the *ModelTransformation$_i$* process. For each element of the *RM$_i$*, there is a part of the *RuleTransformation* process that is in charge of translating the rules into the ATL code, which constitutes the M2M transformation of the *ModelTransformation$_i$* process. Despite the *RuleTransformation* process has been developed in order to adapt architectural models representing user interfaces, this approach is extendable to generate any type of M2M transformation, which is executed on a rule model.

## 4 RELATED WORK

Nowadays there are different proposals to achieve adaptive transformations for architectural models at runtime. To this end, in (Gray et al., 2006) the authors developed meta-transformations as transformations which produce transformations. However, unlike this proposal, in our approach the 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 (Floch et al., 2006), 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 (Serral et al., 2010) the authors used Java modules executed inside an OSGi 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 (Kurtev et al., 2007); 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 (Wagelaar, 2008) 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. The authors suggest creating transformation modules that can be either called from other 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. Thus, we chose to create ATL rules defined by a DSL and dynamically build ATL transformation modules.

On the other hand, in (Tisi et al., 2009) the authors suggested the use of M2M transformations to generate as output transformation models in order to adapt or modify an M2M transformation process. This composition method for transformation process guarantees well-built transformation modules, since we used the ATL metamodel as reference to generate transformation models; however, these *Higher-Order Transformations* (HOT) 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.

The approach developed in (Porres, 2005) 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 instead of UML ones and ATL language rather than Python. We carried out the selection of transformation rules through model transformations.

## 5 CONCLUSIONS

Here we presented our proposal of adaptive transformations for architectural models at runtime. 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 context. Therefore ,the transformation rules define the degree of adaptability of our system; the 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 taking into account, in the selection logic, factors as use frequency of transformation rules, rule weight management policy, etc. We also intend to possibly carry out, through HOT (Tisi et al., 2009), the process by which we turn rule models into transformation processes applied to architectural models. We will 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.

## ACKNOWLEDGEMENTS

## REFERENCES

Blair, G., Bencomo, N., France, R.B.: Models@run.time (Special issue on Models at Run Time). Computer, 40(10):22–27 (2009)

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)

Criado, J., Padilla, N., Iribarne, L., Asensio, J.: User Interface Composition with COTS-UI and Trading Approaches: Application for Web-Based Environmental Information Systems. WSKS'2010, Part I, CCIS 111, pp. 259–266, Springer (2010)

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)

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)

Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. Computer, 39(2):51–58 (2006)

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)

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)

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming, 72(1-2):31–39 (2008)

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)

Porres, I.: Rule-based update transformations and their application to model refactorings. Software and Systems Modeling, 4(4):368–385 (2005)

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)

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. ECMDA-FA, pp. 18–33 (2009)

Wagelaar, D.: Composition techniques for rule-based model transformation languages. ICMT, pp. 152–167 (2008)