A Model-Driven Approach to Graphical User Interface Runtime Adaptation

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

[1] Applied Computing Group, University of Almeria, Spain
{javi.criado,npadilla,luis.iribarne}@ual.es
http://www.ual.es/acg
[2] Department of Information Technology and Communications, Technical University of Cartagena, Spain
cristina.vicente@upct.es
http://www.dsie.upct.es/personal/cristinav/

Abstract. Graphical user interfaces play a key role in human-computer interaction, as they link the system with its end-users, allowing information exchange and improving communication. Nowadays, users increasingly demand applications with adaptive interfaces that dynamically evolve in response to their specific needs. Thus, providing graphical user interfaces with runtime adaptation capabilities is becoming more and more an important issue. To address this problem, this paper proposes a component-based and model-driven engineering approach, illustrated by means of a detailed example.

Keywords: runtime model adaptation, model transformation, graphical user interface

1 Introduction

Graphical User Interfaces (GUIs) play a key role in Human-Computer Interaction (HCI), as they link the system with its end-users, allowing information exchange and improving communication. Nowadays, users increasingly demand "smart" interfaces, capable of (semi-)automatically detecting their specific profile and needs, and dynamically adapting their structure, appearance, or behaviour accordingly.

GUIs are increasingly built from components, sometimes independently developed by third parties. This allows end-users to configure their applications by selecting the components that provide them with the services that better fit their current needs. A good example of this is iGoogle, as it provides end-users with many gadgets, allowing them to create personal configurations by adding or removing components on demand.

Following this trend, the proposal presented in this paper considers GUIs as component-based applications. Furthermore, it considers the components integrating GUIs software architectures at two different abstraction levels: (1) at design time, components are defined in terms of their external interfaces, their

internal components (if any), and their visual and interaction behaviours, while (2) at runtime, the former abstract components are instantiated by selecting the most appropriate Commercial-Off-The-Shelf (COTS) components (i.e., those that better fit the requirements imposed both by the abstract component and by the global GUI configuration parameters) from those available in the existing repositories.

Our proposal does not only rely on a component-based approach but also, and primarily, on a Model-Driven Engineering (MDE) approach. As detailed in the following sections, we propose a GUI architecture description meta-model that enables (1) the definition of component-based abstract GUI models at design-time, and (2) the runtime evolution (be means of automatic model-to-model transformations) of these architectural models according to the events detected by the system. The instantiation of these abstract models at each evolution step is out of the scope of this paper.

The remainder of the article is organized as follows. Section 2 reviews related works. Section 3 describes the proposed approach and its constituting elements, namely: the proposed GUI architecture meta-model, and a set of model transformations enabling runtime GUI adaptation. In order to illustrate the proposal, a GUI model evolution example is also described in detail in this section. Finally, Section 4 draws the conclusions and outlines future works.

2 Related Work

There are many model-driven approaches in the literature for modelling user interfaces, as detailed in [1]. Some of them use a MDE perspective for web-based user interfaces [2]. However, in most cases, models are considered static entities and no MDE technique is applied to add dynamism, for instance, using model transformations.

Model transformations enable model refinement, evolution or even, automatic code generation. In [3], the authors investigate the development of plastic user interfaces (which have the context adaptation ability), making use of model transformations to enable their adaptation. However, these transformations are used at design-time and not at runtime, as we propose here. In [4], the authors propose an approach that makes use of model representations for developing GUIs, and of model transformations for adapting them. This work, in which the research described in this paper is based on, also considers these GUI models as a composition of COTS components.

The adoption of Component-Based Software Development (CBSD) proposals for software applications design and implementation is increasingly growing. An example can be found in [5], where the authors identify the multiple GUI evolution possibilities that come from working with component-based software architectures (e.g., addition of new components, interface reconfiguration, adaptation to user actions or task, etc.). However, this proposal implements GUI evolution by programming GUI aspects, instead of using model transformation techniques, as we propose in this work. Another example is shown in [6], which

presents a combined MDE and CBSD approach to enable the modelling of structural and behavioural aspects of component-based software architectures. However, this proposal is aimed at general-purpose software architectures, and not particularly suited for GUI development. In [7], the authors focus their research on component retrieval, composition and re-usability in *standalone* DSLs (Domain Specific Languages). This is useful in web applications, especially in those making use of the semantic web. However, as before, this work does not apply these ideas directly to compose GUI applications.

On the other hand, recent software engineering proposals advocate for the use of models at runtime (models@runtime) [8]. Existing research in this field focuses on software structures and their representations. Thus, significant bodies of work look at software architecture as an appropriate basis for runtime model adaptation [9]. Our vision of models@runtime is completely aligned with this idea as our GUIs are, in fact, architecture models. In [10], the authors study the role of models@runtime to manage model variability dynamically. Their research focuses on reducing the number of configurations and reconfigurations that need to be considered when planning model adaptations. However, this work is not focused on GUIs, but in Custom Relationship Management (CRM) applications.

Next section presents the proposed GUI modelling and runtime adaptation approach, in which GUIs will be modelled as component-based architectures. These architecture models will be capable of evolving through model transformations in order to self-adapt according to the events detected by the system.

3 Runtime GUI Adaptation

This paper focuses on applications with *Graphical User Interfaces* (GUI). In fact, our application models may contain any number of GUIs (e.g., one for each type of user). Each GUI, in turn, is built by assembling components, in particular COTS, which are well known in the CBSD literature. We call these components *cotsgets* for their similarity to the gadgets, widgets and other components frequently used in GUI development.

All the *cotsgets* included in each GUI, together with their behaviour and the composition and dependency relations that may exist among them, conform the GUI architecture. As we have opted for a MDE approach, we model GUI architectures using a meta-model. This architecture meta-model can be seen as an aggregation of three parts or subsets, namely: (1) an *structural meta-model*, (2) an *interaction meta-model*, and (3) a *visual meta-model*.

Firstly, the *structural meta-model* allows designers to model composition and dependency relationships among components. Dependencies are modelled by connecting component ports, which may provide or require any number of interfaces (sets of services). Secondly, the *interaction meta-model* is used for modelling the behaviour associated with user-level interactions, defined in terms of the performed tasks. This meta-model includes concepts such as roles, subtasks, choreography, etc. Finally, the *visual meta-model* aims to describe the

component behaviour from the point of view of its graphical representation on the user interface.

As a solution to the interface adaptation process, this work proposes a MDE approach to GUI model evolution [11], where interface architectures are considered as models capable of evolving at runtime. To achieve this, we have implemented a two-stage process, where: (1) the interface architecture models, defined in terms of abstract components, are evolved by means of a model-to-model transformation according to the (user or application) events detected by the system, and (2) the resulting abstract models are then instantiated by means of a regeneration process, where a *trader* selects (from the existing repositories) the cotsgets that better fulfill the requirements imposed by the abstract architecture model, and then regenerates the application in terms of the selected executable components. Thus, the first stage of the process (*transformation phase*) deals with the runtime adaptation of the abstract interface architecture models, while the second one (*regeneration phase*) deals with their instantiation (see Figure 1). It is worth noting that this article is focused only on the transformation phase.

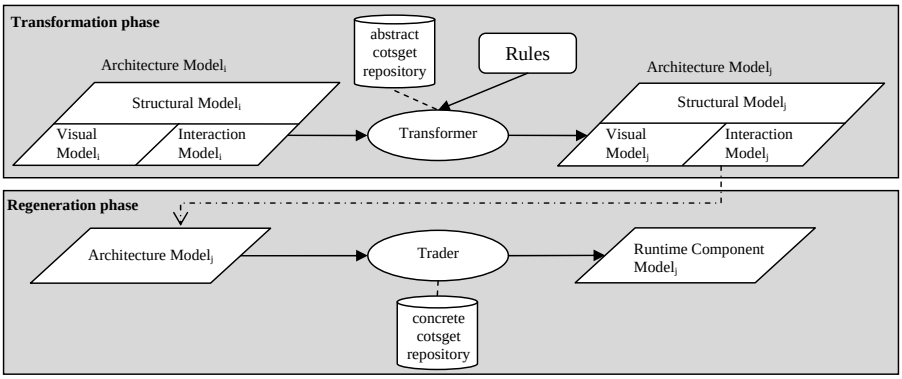

Fig. 1. Schema of Model Adaptation

The model-to-model transformation, implementing the first stage of the process, comprises a set of rules that define how to evolve the current abstract interface architecture model depending on the events detected by the system. As an output, the transformation generates a new abstract interface architecture model, defined in terms of the same meta-model as the input one (i.e., the transformation evolves the input model rather than translating it from one modelling language into another). For the sake of clarity, we have implemented this transformation in two parts: (1) the first one, takes the input interface model and evolves the state machines associated to its components according to the detected event, and (2) the second one executes the actions associated to the new current states of the evolved state machines. Further details about this transformation will be given next in section 3.2.

3.1 Architecture Meta-model

In this paper, we focus on the structural and the visual subsets of the architecture meta-model. The former enables the description of the software architecture in terms of its internal components and the connections existing among them. Similarly, the later enables the specification of the system visual behavior according to the expected runtime adaptation to certain user or application events. An excerpt of the architecture meta-model, showing the main concepts included in these two subsets, is shown in Figure 2.

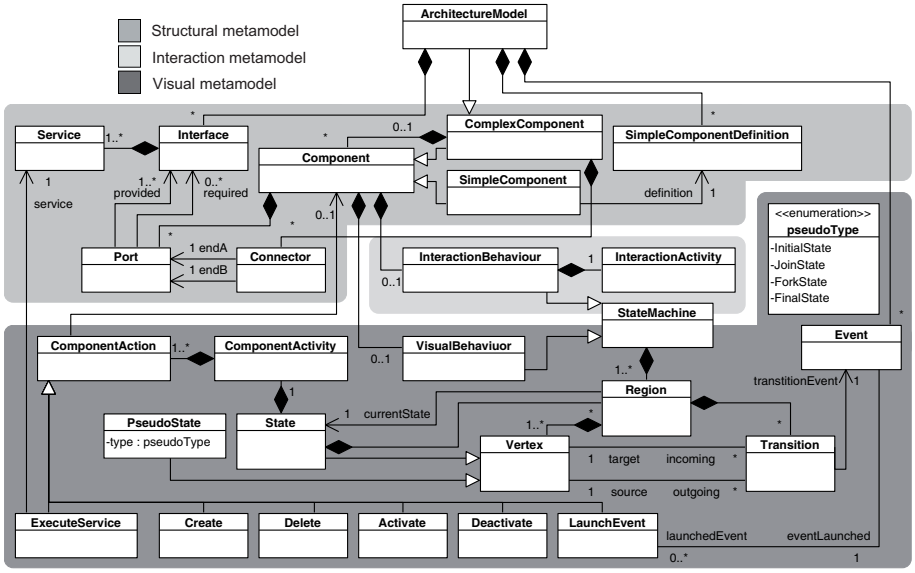

Fig. 2. Architecture Meta-model

The concept *ArchitectureModel* is the meta-model root, and it contains component interfaces (*Interface*), simple component definitions (*SimpleComponentDefinition*), and all the events considered relevant for the system evolution (*Event*). Being defined in the root of the model, these three kinds of elements can be reused by all the other elements in the model. Both *ComplexComponent*s and *SimpleComponent*s are subtypes of the abstract meta-class *Component*. *SimpleComponent*s have a reference to their corresponding *SimpleComponentDefinition*, while *ComplexComponent*s are defined in terms of their internal *Component*s, which can be, in turn, either simple or complex. Each *Component* contains two behavioural descriptions: (1) a *VisualBehavior* which, using a state machine model, defines how each component visually evolves depending on certain *Event*s, and (2) an *InteractionBehavior*, which enables designers to model user's interaction and cooperation (this is out of the scope of this paper).

The *StateMachine*s used to model the component visual behaviour may contain any number of orthogonal (i.e., concurrent) *Region*s which, in turn, may contain any number of *State*s. Each *State* contains a *ComponentActivity* that models the workflow of *ComponentAction*s that need to be executed when the component reaches that state. On the other hand, the *Transition*s between states are associated to one of the *Event*s defined in the ArchitectureModel.

It is worth noting that this work is not intended not prescribe how to construct or deduce the state machine model that better describes each component visual evolution. Conversely, this work is focused on the model transformation supporting that evolution, which implementation is detailed next.

3.2 Runtime Model Adaptation Process

As previously stated, the runtime adaptation of the abstract inteface architecture model has been implemented by means of a model-to-model transformation (Figure 3). This transformation, defined as a set of rules, takes the current interface model (AM_A) and a detected event as its inputs, and generates an evolved interface model (AM_B) as its output. Although the process seems quite straight forward, implementing it in one step is not that easy. Thus, for the sake of simplicity, we have splitted the transformation in two.

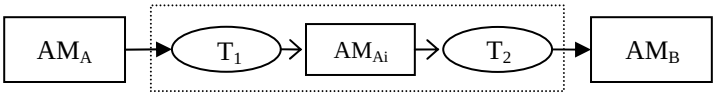

Fig. 3. Adaptation process

The first part of the model transformation (T_1) takes the interface architecture model and the event collected by the system as an input (AM_A), and produces an intermediate interface architecture model (AM_{Ai}) where all the state machines being affected by the collected event are appropriately updated. To achieve this, the transformation finds, for all the *currentState*s (one for each region in every state machines in every component), all the outgoing transitions being fired by the collected event, and updates the *currentState* to the target of the fired transition. Once the state machine models have been updated, the second model transformation (T_2) is exectuted, taking the resulting AM_{Ai} model as an input. The role of this second transformation is to execute the *ComponentAction*s contained in all the updated *currentState*s. As a result, a new interface architecture model (AM_B) is generated. In this first approach to model GUI evolution, we have defined six types of actions that might be executed on a component (as a result of an event launched either by the user or by other component): `Create`, `Delete`, `Activate`, `Deactivate`, `Execute Service` and `Launch Event` (see Figure 2). Table 1 shows two example rules, each one belonging to one of the transformations.

Table 1. Example of ATL rules

T	ATL rule

T_1	`rule RegionEvolution`
	`{ from f: INMM!Region`
	`(f.smParent.parent.parent.currentEvent.eventTransition->exists(t |`
	`t.source = f.currentState) )`
	`to o: OUTMM!Region`
	`(name<-f.name,transitions<-f.transitions,vertex<-f.vertex,`
	`currentState<-f.smParent.parent.parent.currentEvent.eventTransition->`
	`select(t |t.source = f.currentState)->collect(t | t.target),`
	`update<-true) }`

T_2	`rule CreateActionExecutable(f: INMM!Create)`
	`{ to t: OUTMM!Create(parameters<-f.parameters,`
	`sourceComponent<-f.sourceComponent, parent<-f.parent),`
	`c: OUTMM!SimpleComponent`
	`(name<-f.parent.parent.parent.smParent.parent.name +`
	`f.sourceComponent.name,`
	`parent<-f.parent.parent.parent.smParent.parent,`
	`definition<-f.sourceComponent) }`

The first of these example rules (*RegionEvolution*) belongs to T_1 and is related to *Region* elements. This rule only affects those *Region*s containing a transition that (1) has the *Region*'s current state as its source, and (2) is fired by the current event. As a result of apply this rule, the value of the current state will be changed (to the state being the target of the fired transition) and the 'update' attribute will be set to 'true' (to inform the second transformation that the actions associated to that state need to be executed). The second example rule (*CreateActionExecutable*) belongs to T_2 and it is called when the transformation finds a `Create` type action that needs to be executed. Its purpose is to copy the *CreateAction* element to the output model and also to add the *Component* associated to this action (*sourceComponent*) to the interface model.

We have implemented our two model transformations using ATL (ATLAS Transformation Language) [14]. The ATL language is a Domain Specific Language (DSL) aimed at describing model-to-model transformations. ATL is inspired on QVT and is a hybrid language that allows both declarative and imperative constructs. We decided to use ATL as its implementation is quite robust, and it is widely spread in use by the MDE community.

In this first approach, random events are simulted and both transformations are manually launched one after the other. However, we are working on an improved implementation that automatically invokes both transformations every time an event is detected, making use of the ATL facilities for programatically executing transformations.

3.3 A GUI Runtime Adaptation Example

In order to illustrate the proposed approach, this section presents a case study on an example GUI runtime adaptation. It describes in depth a few steps of the adaptation process.

The example shows an interface architecture model composed by two graphical user interfaces (GUI_1 and GUI_2). Each of these GUIs has two simple components (C_1 and C_2). We will simulate an event that adds a *Chat* component to GUI_1. This event will also produce the addition of a *Chat* component to GUI_2. Finally, we will also simulate the generation of a new event that deletes GUI_2 (and all its subcomponents) from the architecture model. Figure 4 shows a snapshot of the interface architecture model at the initial stage ($\texttt{ArchitectureModel}_0$).

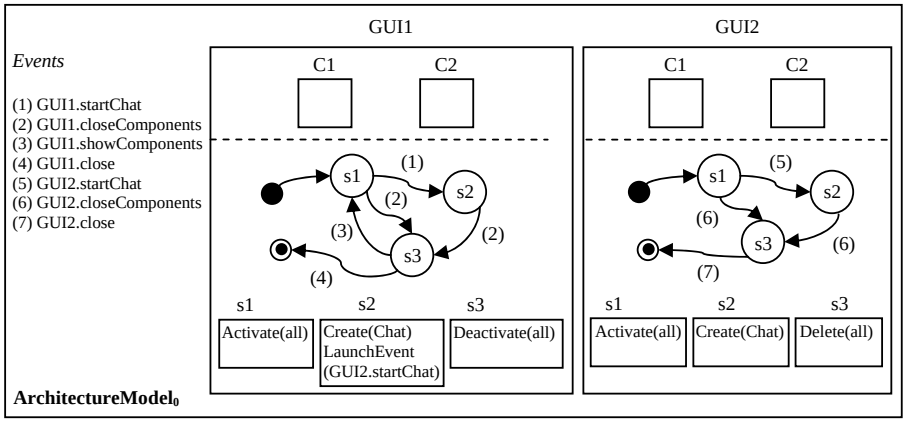

Fig. 4. Initial Architecture Model

Given the initial interface architecture model, when an event occurs the adaptation process starts. An example of this process is shown in Figure 5, which illustrates the model transformation steps executed after the `GUI1.startChat` event happens. As result, the first transformation (T_1) evolves the state machine associated to `GUI1`, changing its current state to `s2`, as indicated by the model. Then, when the T_2 transformation is launched, it executes the `Create(Chat)` and `LaunchEvent(GUI2.startChat)` actions. The first action implies the addition of a `Chat` component within `GUI1`, while the second action causes the launching of a `GUI2.startChat` event. We obtain *Model B* as result.

The adaptation process concludes when all the events haven processed. However, the `GUI2.startChat` event still needs to be attended. Thus, T_1 is launched again and the `GUI2` component changes its current state to `s2`. Finally, T_2 executes the `Create(Chat)` action, resulting in the addition of a new `Chat` component within `GUI2`. In this case, we obtain *Model C* as a result.

Next, figure 6 shows another adaptation example starting from *Model D* (obtained by setting GUI2.closeComponents as the system *currentEvent* in Model C). In this case, we show the models involved in the adaptation process using the reflective model editor provided by the Eclipse Modeling Framework (EMF). In the first step, T_1 changes the current state of `GUI2` to `s3` and sets the 'update'

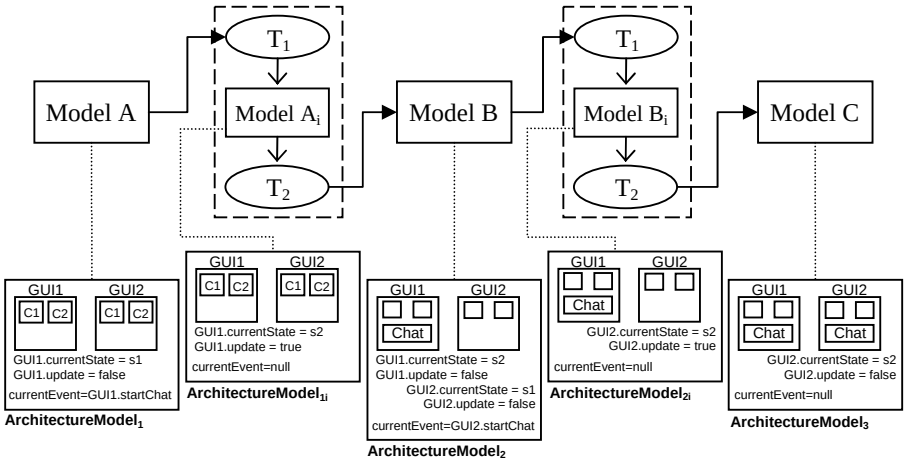

Fig. 5. Transformation Steps

attribute to 'true', as shown in *Model* D_i (central column in figure 6). Then, T_2 executes the `Delete(all)` action, producing the deletion of all the components in `GUI2`, including itself, as shown in *Model E* (right column in figure 6).

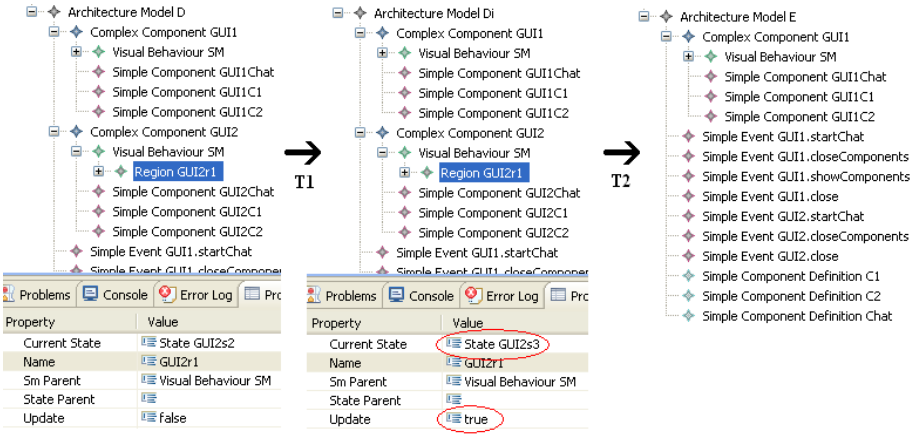

Fig. 6. From *Model D* to *Model E*

4 Conclusions and Future Work

Nowadays, the increasingly growing number and complexity of Information Systems force developers to make them more flexible, easy to adapt and evolve,

and accessible for being manipulated at runtime. Graphical User Interfaces play a key role in this kind of systems, as they ease communication between applications and their end-users. Thus, it is necessary to obtain GUI dynamism and adaptability to user profiles and context. It is also very important to get this adaptation while the system is running, without stopping its execution and without re-modelling its components; in other words, to support a runtime adaptation process. However, most GUIs are still built based on traditional software development paradigms, which do not take into account that they have to be distributed, open, changeable and adaptable. In contrast, GUIs should be able to regenerate themselves at runtime depending on the context, the user interactions, and the changing application requirements.

In this paper, we have presented a MDE approach for the development of adaptable graphical user interfaces. The proposed approach aims to build these applications as assemblies of GUI components. These GUIs will be architecture models that can evolve over time through model transformations with the purpose of changing and adapting on system events. We describe a combined MDE and CBSD proposal to GUI architecture modelling and runtime adaptation. This approach revolves around (1) a meta-model for formally specifying the component-based structure and the visual and interaction behaviour of GUI architectures, and (2) a model-to-model transformation that enables GUI model evolution according to the behavioural rules and the actions performed by the user at runtime. Finally, a runtime adaptation example has been presented that illustrates the proposed approach.

As future work, we would like to develop a graphical tool using the *Eclipse Graphical Modeling Framework* (GMF) [15] in order to easily draw new GUI models conforming to the proposed architecture meta-model. We also plan to evaluate how the use of other MDE tools, in particular those enabling the inclusion of action semantics in meta-models (e.g., Kermeta [16] or AMMA [17, 18]) can help us improving our runtime adaptation process. Besides, we plan to enable the inclusion of alternative behaviours that can be appropriately activated depending, e.g., on previous GUI evolution records or on any relevant contextual information (i.e., enabling the adaptation not only of the interface models but also of the model-to-model transformation itself, also at runtime). Furthermore, we also plan to automate the adaptation process by making use of the ATL facilities for programatically executing the transformations.

Finally, we are interested in studying possible change detection in the interaction meta-model (not covered in this article) by means of automated co-evolution mechanisms and meta-model adaptations [19, 20].

Acknowledgments. This work has been supported by the EU (FEDER) and the Spanish MICINN under grant TIN2007-61497 and TIN2010-15588 project.

References

1. Pérez-Medina, J.L., Dupuy-Chessa, S., Front, A.: A Survey of Model Driven Engineering Tools for User Interface Design. Task Models and Diagrams for User

Interface Design, LNCS 4849, pp. 84–97 (2010)
2. Chavarriaga, E., Macia, J.A.: A model-driven approach to building modern Semantic Web-Based User Interfaces. Advan. in Eng. Soft. 40, 1329–1334 (2009)
3. Sottet, J.S., Calvary, G., Favre, J.M., Coutaz, J., Demeure, A.: Towards Mapping and Model Transformation for Consistency of Plastic User Interfaces. CHI2006, Workshop on The Many Faces of Consistency in Cross-platform Design, (2006)
4. Iribarne, L., Padilla, N., Criado, J., Asensio, J.A., Ayala, R.: A Model Transformation Approach for Automatic Composition of COTS User Interfaces in Web-Based Information Systems. Information Systems Management, 27, 207–216 (2010)
5. Grundy, J., Hosking, J.: Developing adaptable user interfaces for component-based systems. Interacting with Computers, 14, 3, 175–194 (2002)
6. Alonso, D., Vicente-Chicote, C., Barais, O.: V3Studio: A Component-Based Architecture Modeling Language. In 15th IEEE Conf. ECBS, pp. 346–355, (2008)
7. Henriksson, J., Aßmann, U.: Component models for semantic web languages. Semantic techniques for the web, 233–275 (2009)
8. Blair, G., Bencomo, N., and France, R.B (eds.): Models@Run.Time. Special Issue, Computer, IEEE Computer Society (2009)
9. Garlan, D., Schmerl, B.: Using architectural models at runtime: Research challenges. Software Architecture, pp. 200–205 (2004)
10. Morin, B. and Barais, O. and Jézéquel, J.M. and Fleurey, F. and Solberg, A.: Models at Runtime to Support Dynamic Adaptation. IEEE Computer, 42(10), pp. 44–51 (2009)
11. Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution. Software Evolution, pp. 1–11 (2008)
12. ISO, *Information Technology — Open Distributed Processing — Trading Function: Specification*. ISO/IEC 13235-1, ITU-T X.950
13. Iribarne L, Troya JM, and Vallecillo A: A Trading Service for COTS Components. The Computer Journal. 4, 3, pp. 342–357 (2004)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming. 72 (1–2), 31–39 (2008)
15. Eclipse Fundation. Eclipse Graphical Modeling Framework (GMF), `http://www.eclipse.org/modeling/gmf`
16. Muller, P.A. and Fleurey, F. and Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. Model Driven Engineering Languages and Systems, pp. 264–278 (2005)
17. Del Fabro, M.D. and Jouault, F.: Model transformation and weaving in the AMMA platform. In Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE), Braga, Portugal, pp. 71–79 (2005)
18. Di Ruscio, D. and Jouault, F. and Kurtev, I. and Bézivin, J. and Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Laboratoire D'Informatique de Nantes-Atlantique. Research Report (2006)
19. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. 12th Int. IEEE EDOC, pp. 222-231 (2008)
20. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. ECOOP 2007, LNCS 4609, pp. 600–624 (2007)