View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by idUS. Depósito de Investigación Universidad de Sevilla

# An MDE approach for Runtime Monitoring and Adapting Component-based Systems: Application to WIMP User Interface Architectures

Javier Criado, Luis Iribarne, Nicolás Padilla
*Applied Computing Group*
*University of Almería, Spain*
{*javi.criado, luis.iribarne, npadilla*}@*ual.es*

Javier Troya, Antonio Vallecillo
*GISUM/Atenea Group*
*University of Málaga, Spain*
{*javiertc, av*}@*lcc.uma.es*

***Abstract*—In certain systems, software must be adapted at runtime to the requirements and changes occurring in the context. A strategy to achieve this goal is to model such systems as software architectures making use of the Component-based Software Engineering (CBSE). Thus, the system can be adapted through the reconfiguration of the software architectures. In this paper we present a schema for the adaptation of software architectures at runtime based on the system context observation. The software system is defined by means of architectural models at two levels: abstract and concrete. We use a trading process to regenerate concrete architectural models from their abstract definitions and a component repository. We also use Model-Driven Engineering (MDE) techniques to transform at runtime such models in order to achieve the system adaptation to the monitored context by using observers. This article describes a case study of component-based user interfaces to illustrate our approach.**

***Keywords*-MDE, adaptive transformation, observer, trading**

## I. Introduction

Nowadays, many software systems need to self-adapt according to changes in their execution environment, as changes in the values of context variables, changes in user interaction with the system, or changes due to external entities [1]. Ideally, these systems should self-adapt at runtime with as little human intervention as possible. Furthermore, these systems normally have a lot of information and it is very complex to define runtime adaptation mechanisms. The idea, then, is to develop adaptation mechanisms that leverage software models, what is referred to as models@runtime. It uses the concepts in *Model-Driven Engineering* (MDE) and extends them with runtime capabilities. MDE aims to raise the level of abstraction in program specification and increase automation in program development. It proposes to use models at different levels of abstraction for developing systems. The use of executable model transformations increases automation in program development. In this way, *higher-level* models are transformed into *lower-level* models until the model can be made executable by using either code generation or model interpretation.

Regarding component-based architectures [2], MDE tools and techniques play a key role in their design and development. Furthermore, experience is showing that MDE can be even more effective for architectural model generation at runtime [3]. Particularly, it is possible for different final software architectures to be generated at runtime from the same *abstract* specifications, according to end-user context properties such as platform, user roles, component states, etc. In this context it is important to consider variability mechanisms that provide the appropriate levels of adaptability required to dynamically adapt models at runtime.

The proposal presented in this paper focuses on the adaptation of component-based systems at runtime, represented as architectural models. These models contain the specification of the components making up the architecture [4], which combined together provide the required software functionality. Within our system, we distinguish between two levels of abstraction: the *abstract* level and *concrete* level. The former defines the component types (in addition to their specifications) to be included in the software architecture, while the later contains references to concrete components (within the repository) that will form part of the final architecture. Thus, the architectural adaptation is carried out by two processes: a *transformation* process of the abstract definitions, followed by a *regeneration* process at the concrete level [5].

The *transformation* process aims to enable the evolution and adaptation of abstract architectural models. We follow an MDE methodology so that we can achieve their change and adaptation by using model-to-model transformations (M2M). A transformation definition is composed by a set of transformation rules that together describe how a source model can be transformed into a target one. A transformation rule, in turn, is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. At the concrete level, the *realization* of the software architecture is achieved by a trader [6] that looks into existing repositories of concrete components for those fulfilling the requirements imposed by the abstract architectural model. The trader selects the right set of components for the application. Every time a new abstract architecture is identified (normally due to changes in the user requirements or in the running environment), the trader again finds the suitable components that realize it.

The main contribution of this paper is a new mechanism of adaptation, provided by the use of *observer* objects that monitor the state and behavior of the components accomplishing the software architecture (*i.e.*, monitoring the concrete architectural model). In our proposal, observers are used to trigger the model transformations that perform the adaptation process. A second, and more interesting, use of observers is to trigger a *lower-level* adaptation process whereby the abstract architecture does not need to be changed, but only one of its realizing components.

As experimental example scenario, we are interested in modeling of simple and *friendly* UIs *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 (UI) are described by means of architectural models that contain the specification of UI components. 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.

The rest of the article is organized as follows. Section II describes the main scenario of the proposal. Section III explains the adaptation process. Section IV presents an example of adaptation of a component-based user interface architecture. Section V reviews related work. Finally, Section VI outlines the conclusions and future work.

## II. Background scenario

Globalization of information and the knowledge society on the Internet requires the modernization of Web-based Information Systems (WIS). This is ready to be easily adaptable, extensible, accessible and manageable at runtime by different people and/or groups of people with common interests. Special attention has been given to globalization of information through a common system vocabulary using ontologies and web semantics. However, WIS user interfaces are still being constructed on the basis of traditional software development paradigms, without taking into account in their construction globalization issues such as distribution, opening and changes. This means that a WIS UI must be able to be dynamically reconstructed at runtime depending on the type of interaction (individual or collective) and the purpose of the interaction (management, technical, etc.).

Under this scenario, our interest is focused on studying and developing an experimental methodology to solve the self-adaptation problem of user interfaces on *Web-based Environmental Information Systems* (a kind of WIS) [7]. The experimental methodology initially works with simple and *friendly* WIMP user interfaces (Windows, Icons, Menus and Pointers) [8]. Such user interfaces are based on "bottom-up" composition at runtime of *widgets*-type COTS interface components. The methodology allows studying scenarios for the interaction of evolutive and cooperative user interfaces. In the methodology user interfaces are considered as architectures made-up of *widgets*-type components. This sort of architectures respect some principles of composition, e.g., dependence between components, restrictions in use, availability and visibility, etc.

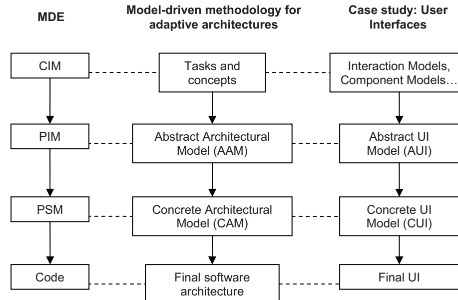


Figure 1. Model-driven methodology

Our proposal aims to structure the development life cycle of component-based systems into four levels of abstraction, from the task specification to the running software architectures (Figure 1). The **Task and concepts** level matches the CIM (*Computational-Independent Model*) level in MDE, it represents the tasks that need to be performed in order to reach the system requirements and the domain objects manipulated by these tasks. The **Abstract Architectural Model (AAM)** level corresponds to the PIM (*Platform-Independent Model*) level in MDE and is the abstract definition of a software architecture. It represents the architecture in terms of what kind of components it must contain, how the relationships between them are, and what specifications these components have. The **Concrete Architectural Model (CAM)** level corresponds to the PSM (*Platform-Specific Model*) one in MDE and is the concrete definition of a software architecture. It describes which concrete components (available in the repository) best fulfill the abstract definition of the architecture. Finally, the **Final software architecture** level corresponds to the code level. It is made up of the source code which will be interpreted or compiled, generating in this way the running software system.

As we advanced, WIMP user interfaces will represent an example of component-based architecture in our methodology. As in [9], our models of UI components, of interaction, etc., correspond to the CIM level. The *abstract* UI would be at the PIM level, and the *concrete* UI would be found at the PSM one. The final UIs shown to the users would be in the code level in MDE. Figure 2 shows an example of a graphical UI that describes the four levels of the methodology. In this UI example a user needs a communication task which requires the use of chat, and some communication via audio and video. The *abstract* architecture is an AAM model containing the `Chat`, `Audio` and `Video` abstract components, and the *concrete* architectural model and its final software architecture could be the ones in Figure 2.

According to the example, AAM is offered to the *SemanticTrader* to calculate the configuration of *concrete* compo-

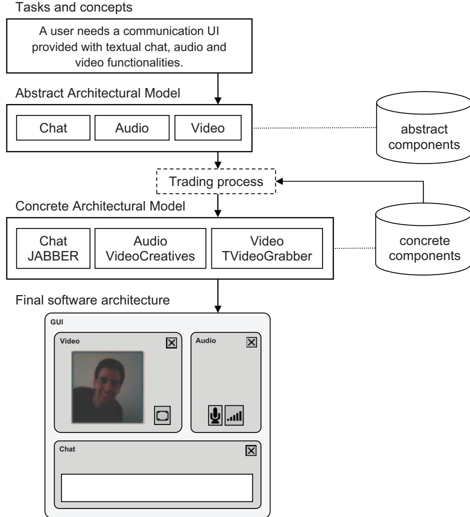


Figure 2. A component-based UI architecture example

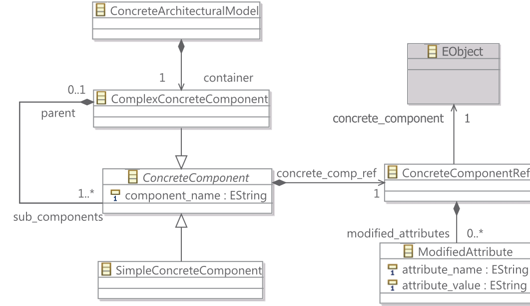


Figure 3. CAM metamodel

nents that best meets the abstract definition, thus generating a concrete architectural model (CAM). Let us suppose for this example, among all possible configurations, the trader chooses the `ChatJABBER`, `AudioVideoCreatives` and `VideoTVideoGrabber` concrete components, which realize the `Chat`, `Audio` and `Video` abstract components, respectively (Figure 2). The concrete architectural model generated by the *SematicTrader* process conforms to the meta-model in Figure 3. In this way, CAM is made up of `ConcreteComponent` elements whose type could be simple or complex. Both types contain a reference to the corresponding concrete component, which is located in the concrete component repository model. `ConcreteComponent` elements can also contain information about whether any of their attributes does not have the default value (`ModifiedAttribute`) that their specification marks.

Throughout the development of this research work, we decided to separate the abstract and concrete levels of our component-based systems as well as the models that represent the component repositories and those relating to information monitored by observers. This has been adopted to facilitate the system design as well as for monitoring and processing the adaptation at runtime [10].

## III. Adaptation process

As explained above, the architectural model adaptation is achieved by using a two-stage process: a transformation phase, which is focused on the abstract definitions of the architectural models, and a regeneration phase focused on the concrete definitions of them. This paper describes the second phase in detail, which is focused on the concrete architectural model level by means of an observer model.

### A. Observer models

Our goal is to obtain adaptive architectures capable of adapting to the context. To achieve this automatic response, it is necessary to monitor those elements within the context that can be monitored, and those elements whose change generates the need for adapting the architecture. In order to achieve this monitoring, we have included (in the meta-model describing the component properties) the possibility of specifying that an attribute of a component is "observable" through an external process. It is also necessary to link the observable properties of the components with those requirements of the system they affect [11]. In this way, if the value of some of these observed variables changes, it will be possible to ascertain if the requirements are still met and, if the architecture does not currently satisfy the requirements, the system must be able to determine which type of adaptation is needed to be performed (Sections III-B and III-C).

Figure 4 shows the DSL that illustrates the relationship between observers and the context variables related to the properties being monitored. The meta-model defines three sorts of observers: `ComponentObserver`, `ObserverObserver` and `ContextObserver`. The first type is intended to monitor the state of the components running in the architecture. This element has a reference to an `EObject` in order to be linked to the corresponding component of the CAM model. The second one aims to gather information about several observers, and the third one is responsible for storing the monitored information of the context variables. Therefore, for each concrete architectural model (CAM), the system generates an observer model (OBM) which stores information about the attributes that are being monitored (both about the components and the context). Thus, let us suppose an architecture with Chat, Audio and Video components, where the video and audio components have an "observable" property associated to the *bandwidth* (i.e., rate of data transfer). Furthermore, the context variable related to the system available *bandwidth* is also being monitored. As a result, the observer model generated for this architecture has three observers: ObVideoBandw,

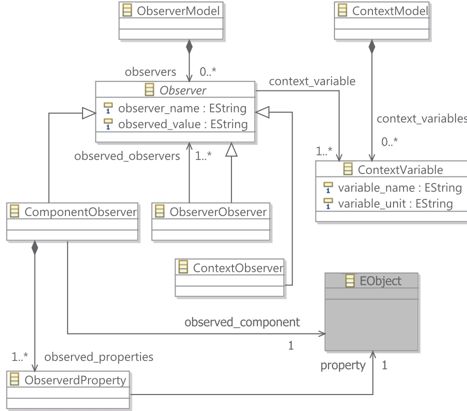


Figure 4. The observer metamodel

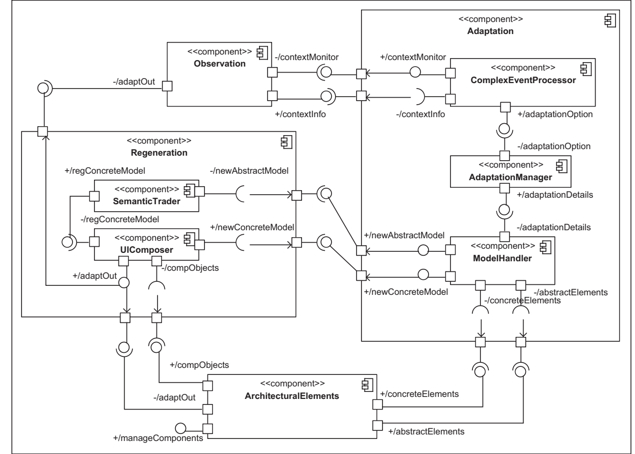


Figure 5. The adaptation schema

related to the video component; ObAudioBandw, linked to the audio component; and ObContextBandw, monitoring the *bandwidth* context variable.

### B. Adaptation Architecture

The system has a model for representing the information being monitored. Apart from this, it needs to have a monitoring mechanism as well as an adaptive mechanism to make architectural changes depending on the changes occurred in the context. For building our adaptive schema, we have chosen to separate the *observation* and the processing of the observed information. Thus, there are three main parts of the adaptation schema encapsulated in three complex components (*Observation*, *Adaptation*, *Regeneration*). These components communicate with another one named *ArchitecturalElements*, which manages the architectural elements and repositories of the system (Figure 5).

Each software component of the final architecture will have an observer component associated whenever there is an element *observer* in the OBM related with that component in the concrete architectural model (CAM). As several changes could occur simultaneously in the observed variables, the observation is centralized by the element called `ObservationManager`, which is a subcomponent of the *Observation* subsystem. Then, the *Observation* component is responsible for providing the *Adaptation* complex component with the changes produced in the context. It is within this component, where a subcomponent named `ComplexEventProcessor` has the task of processing the observed changes and determining if the new values satisfy the system requirements. If they are not fulfilled, the software architecture needs to be adapted and the adaptation options will be provided to the `AdaptationManager`.

### C. Adaptation types

The `AdaptationManager` component can start three different types of adaptation executions, depending on the information provided by the `ComplexEventProcessor` and the existing components in the current architecture:

*1) Modify an attribute of an existing component:* This operation is performed when there is a concrete component of the architecture that is configurable and whose change in the value of some of its editable attributes will imply again the fulfillment of the requirements (taking into account the new observed values of the context variables). This operation affects the CAM. It involves the **ModelHandler** component, which executes an M2M transformation called *ConcreteModelTransformation* (explained in Section IV-B) in order to modify the value of the attribute.

*2) Replace a component:* A replacement operation is needed when a concrete component of the architecture does not fulfill its function properly; for instance, there is an error in the object implementing the component, or the context variation that generates the concrete component does not meet the current requirements. This operation affects the CAM and Observer Model (OBM). This kind of operation involves the use of the component **ModelHandler**, which executes the *ConcreteModelTransformation* with the aim of replacing a concrete component of the architecture and establishing the new associated elements in the observer model (*e.g.*: the new concrete component has an additional observable property).

*3) Delete a component:* This operation is executed when the changes in the context variables produce a breach of system requirements. This can be solved by removing a certain component that implements properties or features that are not mandatory for the proper system operation. For example, in a communication between two system users, a chat component should be offered, but audio and video are optional components. If the bandwidth decreases, the video or audio component can be eliminated, without the communication being completely disrupted. In this case, the operation affects AAM, CAM, and OBM. The operation also involves the

use of the component **ModelHandler**, which executes the *ConcreteModelTransformation* to remove the component of the concrete architecture. The `ModelHandler` component also modifies the abstract architectural model in order to remove its associated abstract component.

*4) Perform a greater change in the architecture:* In this adaptation option, the context changes do not simply require the replacement or removal of a component. These are cases in which there is a breach of the mandatory properties of the architecture as required by the system. It is, therefore, necessary to perform an architectural reconfiguration. The influence operation also affects AAM, CAM and OBM. In this case, the operation involves two components, **ModelHandler** and **SemanticTrader**. The **ModelHandler** component executes a M2M transformation called *AbstractModelTransformation*. It adapts the abstract definition of the architecture depending on the context and the system requirements. It also determines which new components must be inserted in the architecture, which ones will be removed and how the interconnection between them should be. The behavior of this M2M transformation will not be explained in this work because here we focus on the adaptation process whereby the abstract architecture does not need to be recalculated. On the other hand, the **SemanticTrader** component takes as input the AAM model, generated by the *AbstractModelTransformation*, and generates the corresponding CAM model as output. The CAM is calculated from the AAM, the concrete components available in the system, the context variables and the system requirements.

The features of each component (in terms of functional and non-functional properties) are described in the component repositories and are accessible by M2M transformations, the trading process and the `AdaptationManager` component. Thus, within the adaptation schema, the system is able to determine what type of operation is needed, in addition to selecting the components that best fulfill the requirements according to context.

Regardless of the adaptation option, from the new concrete architectural model generated, there will be an adaptation component responsible for generating code or interpreting the model to make up the final software architecture. In our case, since the application domain is that of user interfaces, there is a component called `UIComposer` responsible for showing the final UIs.

## IV. ADAPTATION EXAMPLE

In order to illustrate the adaptation process, we will start from the component-based architecture shown in Figure 2. Please, note that in this paper we will use a very simple example to explore better the adaptation process using observer models. Therefore, let us assume that a system user has an interface providing the functionality of chat, audio and video communication. Due to such communication task, the system must take into account the context variable concerning the available *bandwidth*. Furthermore, it is necessary to explain that the system **requirements** are: (a) The interface should enable the communication between users; (b) Communication should be as stable as possible; (c) Both the interface and the communication should be as comprehensive as possible. Otherwise, related to the system **context**, the available bandwidth is 2 Mbps. Therefore, in the abstract architectural model (AAM) it is specified that the presence of `Chat` component is mandatory, while the `Audio` and `Video` components are optional. The related concrete architectural model (CAM) realized by the trader is the one shown in Section II.

### A. Monitoring the context

Within the CAM, the audio and video concrete components have an "observable" attribute related with the system *bandwidth*. Thus, the observer model (OBM) corresponding to this CAM has two observers of type `ComponentObserver` (see Figure 4), ObAudioBandw and ObVideoBandw, which are in charge of monitoring the use of the bandwidth from the audio and video components. The OBM, in turn, contains another observer named ObContextBandw which is a `ContextObserver` element that monitors the state of the context variable representing the available bandwidth.

Considering the above, let us suppose that the ObContextBandw observer detects a context change in which the value of the available bandwidth decreases. As a result, a notification of the context change is generated within the *Observation* component of the adaptation schema (Figure 5). Then, the *Observation* component provides the `ComplexEventProcessor` with the monitoring data. Such component is responsible for examining the new context values and determines the adaptation option that should be carried out. Let us suppose that the available bandwidth goes down to a value at which the current concrete component (VideoTVideoGrabber) no longer meets the requirements since the needed bandwidth for normal operation is higher than the available bandwidth, but the value of the bandwidth is not so low as to have to remove the video component. That is, the adaptation required by the architecture does not require a change in its abstract definition (eliminating the abstract video component or so on), but it only has to modify the concrete video component to meet the input requirements of the communication task.

Consequently, the `ComplexEventProcessor` determines whether it is necessary to modify or replace the concrete video component. The `AdaptationManager` checks if the current concrete video component has a configuration option reducing the used bandwidth, that is, if it has some "editable" property related to the bandwidth context variable that provides a value that is smaller than the current one. As the VideoTVideoGrabber concrete component does not have that configuration option, the

AdaptationManager searches the concrete component repository and resolves that the adaptation option to be executed consists of replacing the existing video component by the VideoVideoLab one. Next, the ModelHandler performs the *ConcreteModelTransformation*.

### B. Concrete Model Transformation

This M2M transformation is executed whether any of the following adaptation options occurs: (a) **modify** an attribute of an existing component, (b) **replace** a component, or (c) **delete** a component. It is a M2M transformation of MIMO (multiple-input multiple-output) type taking the concrete architectural model ($CAM_i$) and the corresponding observer model ($OBM_i$) as its input, and generating the new concrete architectural model ($CAM_{i+1}$) and the updated observer one ($OBM_{i+1}$) as output.

Since the whole proposal is developed within an MDE framework, the kinds of operations to be executed by the M2M transformation have been also described by an operation model (OpM). This model is solved by AdaptationManager and it is another input to the *ConcreteModelTransformation* (CMT). The OpM models are built conforming to a DSL which contains the three possible sorts of operations for the three types of adaptation performed by this CMT transformation: Modify, Replace or Remove. The three types have in common an attribute indicating the component affected by the adaptation operation. In addition, the Replace operation model contains the name of the new concrete component to be inserted. A Modify operation, in turn, describes the attribute to be modified and also the new value it takes. Hence, since the adaptation option to be executed is to **replace** the VideoTVideoGrabber video component by the VideoVideoLab one, the $OpM_i$ will be the one shown in Figure 6.

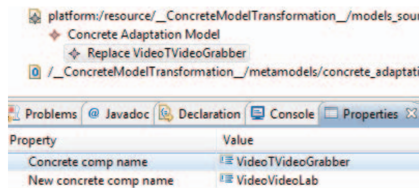


Figure 6. The operation model for the concrete adaptation

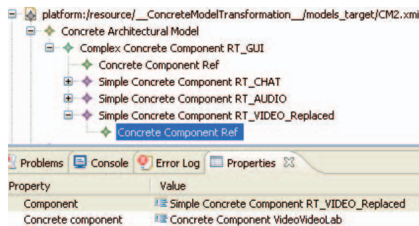


Figure 7. A piece of the concrete architectural model generated by CMT

Table I
TWO *CM* TRANSFORMATION RULE EXAMPLES

```
helper def : isReplaceType() : Boolean =
 OpMM!ConcreteAdaptationModel -> allInstances() ->
 first().operation.oclIsTypeOf(OpMM!Replace);

rule ReplaceConcreteComponent{
 from f: CAMM!SimpleConcreteComponent(
  thisModule.isReplaceType() and
   (f.concrete_comp_ref.concrete_component.component_name =
   thisModule.getOpConcreteCompName())
 )
 to sccomp: CAMM!SimpleConcreteComponent(
  component_name <- f.component_name + '_Replaced',
  concrete_comp_ref <- ccref
 ),
 ccref: CAMM!ConcreteComponentRef(
  concrete_component <- thisModule.concreteComponentRef(
   thisModule.getOpNewConcreteCompName())
 )}

rule ReplaceObservedConcreteComponent{
 from f: OBMM!ComponentObserver (
  thisModule.isReplaceType() and (f.observed_component.
   component_name = thisModule.getOpConcreteCompName())
 )
 to compob: OBMM!ComponentObserver(
  observed_component <- thisModule.concreteComponentRef(
   thisModule.getOpNewConcreteCompName()),
  observed_value <- '-',
  observer_name <- f.observer_name,
  observed_properties <- thisModule.getNewProperties(
   thisModule.getOpNewConcreteCompName()) -> collect (
   p | thisModule.CreateObservedProperty(p)),
  context_variable <- thisModule.getNewContextVariables(
   thisModule.getOpNewConcreteCompName())
 )}
```

The *CMT* transformation is therefore a parameterized M2M transformation in which the parameter is the OpM model. The ATL rules executed in this transformation check the operation type (modify, replace or remove) and the operation content, and execute their actions accordingly. In the case of our example, the transformation rules associated with the replacement of the concrete component and the ones related with the creation of monitoring elements must be performed. In Table I, we can see two example rules for the replacement operation. The new concrete component replaces the old one by changing its name and updating the reference to the concrete component of the repository. On the other hand, the concrete component reference of the corresponding observer element is updated and ObservedProperty elements are created for each observable property of the new component. The resulting concrete architectural model ($CAM_{i+1}$) of the *CMT* is shown in Figure 7. An updated observer model is also generated as output. This $OBM_{i+1}$ contains an updated ComponentObserver element associated to the VideoVideoLab component and it is made up of two ObservedProperty elements related to its two observable properties.

The consistency of this M2M transformation at the concrete level is based on the following assumptions: the concrete source model is correct and well-formed; the abstract definition of this model does not change (in the case of the replacement of a component or the modification of an attribute of an existing component); or the change in the abstract definition, in the case of removing a component, has no consequences to the other components of the architecture

(because there are no dependencies). This verification is performed by the `AdaptationManager` component before the invocation of the *CMT*, so that the transformation should only check the OCL constraints in order to generate a correct target model.

## V. Related Work

There already exist a lot of approaches that present an architecture for systems dealing with models at runtime. They normally use techniques like model-driven engineering (MDE), aspect-oriented modeling (AOM) and component-based architectures (CBA). Different approaches have been proposed to address different problem domains. In [12], the authors apply models at runtime for autonomic reconfiguration of mass-production environments such as those used to create cars or houses, where production costs are a major constraint. Concretely, they focus on the case of smart homes. They present how to achieve autonomic behavior by leveraging variability models at runtime. They use variability models and a dynamic product-line architecture and argue that a system can activate or deactivate its own features dynamically at runtime by fulfilling certain context conditions. In this way, it is these conditions triggering that start the adaptation. In our approach, such adaptation is initiated, at first, by our observers.

In [13], the authors based on aspect oriented modeling in a dynamic software product line which derives products that can be adapted at runtime in order to dynamically fit new requirements or resource changes. The main difference with our proposal is that we accomplish the adaptation transforming the architectural model at the concrete level instead of dynamically weaving the architectural aspects. The authors in [14] also present a runtime architecture to support dynamic software product lines, and they particularly focus on taming the explosion in the number of artifacts while providing a high degree of automation and validation. They combine model-driven and aspect-oriented techniques. In this way, they refine features as aspect models. As we do, they use a model-driven approach and define several meta-models (five in this case) with which the components are specified. Each of these five architectural components has a clear role and well-defined interactions with the other components, as in our approach.

The approach in [15] focuses on models dealing with non-functional properties, such as reliability and performance, and it also presents a case study based on Web-service compositions. They claim that models for non-functional properties should coexist with the implementation at runtime. In this way, automatic checking of the desired requirements is performed while the system is running. However, this approach can only deal with model evolution by continuous estimation of its numerical parameters, but it cannot perform more complex modifications to the model at runtime, such as structural changes. Our proposal also allows defining adaptation rules associated with non-functional properties of the architectural components. In our case, these rules are defined in M2M transformations. Moreover, our trading process is responsible for resolving the optimal concrete configurations of the software architecture taking into account both the functional and non-functional properties.

In [16], the authors present a models at runtime approach based on aspect-oriented and model-driven engineering in the context of mobile computing environments applications that need to dynamically discover services from a wide range of options that may be unknown during design. The specific AOM technique they use is the SMARTADAPTERS approach [17], which has formerly been applied to Java programs and UML class diagrams. In our case, we focus our approach for implementing component-based software architectures; specifically, we have chosen user interfaces as the application domain. Another interesting approach is described in [18] where software adaptation is carried out through MDE and components. The authors use UML profiles to describe the components and their behaviors. This approach named MOCAS relies on behavioral adaptation; in contrast, our approach is based on architectural reconfigurations to get the system adaptation.

Other than this, the concept of *Observer* is not new. Many proposals define it for monitoring the execution of systems and reasoning about some of its properties. In fact, the OMG classifies different kinds of observers in its MARTE specification [19]. As an example, they define *TimedObservers* as conceptual entities that describe requirements and predictions for measures defined on an interval between a pair of user-defined observed events. They must be extended to define the measure that they collect (e.g., latency or jitter) and aim to provide a powerful mechanism to annotate and compare timing constraints over UML models against timing predictions provided by analysis tools. In this sense, they are similar to our observers. The advantage of incorporating them into DSLs by using our approach is that we cannot only use them to describe requirements and constraints on models but we also reason about their behavior. In addition, we can use our observers to dynamically change the system behavior, establishing adaptation rules from the component monitoring, in contrast with the more “static” nature of MARTE observers.

## VI. Conclusions and future work

In this paper we have shown a schema for the adaptation of component-based systems at runtime based on the context observation. The proposal focuses on architectural models, which are defined in two levels of abstraction: the abstract level and the concrete level. The first one defines the component types to be included in the software architecture. The second level contains references to concrete components that will form part of the final architecture.

Therefore, the architectural adaptation is performed by two processes: a *transformation* process of the abstract definitions, followed by a *regeneration* process at the concrete level. The *transformation* process enables the evolution and adaptation of abstract architectural models. We follow an MDE methodology to achieve their change and adaptation by using M2M transformations. At the concrete level, the *regeneration* of the software architecture is achieved by a trader looking up into existing repositories of concrete components for those fulfilling the requirements imposed by the abstract architectural model. The trader selects the right set of components for the application and every time a new abstract architecture is identified (due to changes in the requirements or in the environment), the trader calculates the new suitable components that realize it.

As an advantage (*pros*), we have presented an adaptation schema, where *observer* elements are aimed to monitor the state and behavior of the components accomplishing the software architecture. In this way, observers are used to trigger the model transformations for the adaptation process. A more important goal of our observers is that they can be used to trigger a lower-level adaptation process whereby the abstract architecture does not need to be changed, but only the concrete specification. This proposal is applicable to a wide range of application domains, if they can be modeled as component-based software architectures. As an example domain, we have chosen the field of the “user interfaces”, so we show an adaptation example of a UI in which a concrete component is replaced. On the other hand (as a *cons*), this approach has also a number of limitations. Mainly, M2M transformations in abstract and concrete levels, as well as the trading process that realizes the concrete UIs, add a computational cost that must be taken into account with regard to system performance.

As future work, we intend to build a wide repository of user-interface components, thus we will be able to study a broad range of adaptation scenarios. Moreover, we will provide our trader realizing the concrete architectures with a more powerful *heuristic* taking into account all the possible properties of the components in relation to the context variables impacting on the system. Additionally, we aim to add *traceability* mechanisms in order to inspect the changes occurred in the architectural models during some time. *Variability* of architectural models can also be investigated to study the adaptation and its possible improvements. Finally, we want to investigate the *performance* of the approach to define a more complete evaluation using model checking.

## Acknowledgment

This work has been supported by the EU (FEDER) and the Spanish Ministry MICINN under grant of the TIN2010-15588, TRA2009-0309 and TIN2008-03107 projects, and under a FPU grant (AP2010-3259), and also by the Junta Andalucía under grant of the project TIC-6114.

## References

[1] B. Cheng, R. de Lemos, H. Giese *et al.*, “Software engineering for self-adaptive systems: A research roadmap,” *Software Engineering for Self-Adaptive Systems*, pp. 1–26, 2009.

[2] I. Crnkovic *et al.*, “A classification framework for software component models,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.

[3] N. Bencomo and G. Blair, “Using architecture models to support the generation and operation of component-based adaptive systems,” *Software Engineering for Self-Adaptive Systems*, pp. 183–200, 2009.

[4] L. Iribarne, N. Padilla, J. Criado, J. Asensio, and R. Ayala, “A Model Transformation Approach for Automatic Composition of COTS User Interfaces in Web-Based Information Systems,” *Inf. Syst. Manage.*, vol. 27, no. 3, pp. 207–216, 2010.

[5] J. Criado, C. Vicente-Chicote, L. Iribarne, and N. Padilla, “A Model-Driven Approach to Graphical User Interface Runtime Adaptation,” in *Models@RT*. CEUR-WS, 641:49–59, 2010.

[6] L. Iribarne, J. Troya, and A. Vallecillo, “A Trading Service for COTS Components,” *The Comp. J.*, 47(3):342–357, 2004.

[7] L. Iribarne *et al.*, “The iSOLERES framework,” Spanish Ministry, TIN2010-15588. http://www.ual.es/acg/soleres.

[8] J. Almendros-Jiménez and L. Iribarne, “An extension of uml for the modeling of wimp user interfaces,” *J. Visual Lang. and Computing*, vol. 19, no. 6, pp. 695–720, 2008.

[9] J. Cantera-Fonseca, J. González-Calleros, G. Meixner *et al.*, “Model-Based UI XG Final Report,” May 2010.

[10] T. Vogel and H. Giese, “Adaptation and abstract runtime models,” in *ICSE 2010*, pp. 39–48. ACM, 2010.

[11] J. Troya, J. Rivera, and A. Vallecillo, “On the specification of non-functional properties of systems by observation,” *Models in Software Engineering*, pp. 296–309, 2010.

[12] C. Cetina, P. Giner, J. Fons, and V. Pelechano, “Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes,” *Computer*, 42(10):37–43, 2009.

[13] C. Parra, X. Blanc, A. Cleve, and L. Duchien, “Unifying design and runtime software adaptation using aspect models,” *Science of Computer Programming*, 2011.

[14] B. Morin, O. Barais *et al.*, “Models@RT to Support Dynamic Adaptation,” *Computer*, 42(10):44–51, 2009.

[15] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, “Model Evolution by Run-Time Parameter Adaptation,” in *ICSE 2009*, pp. 111–121. IEEE CSP, 2009.

[16] B. Morin, F. Fleurey, N. Bencomo *et al.*, “An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability,” in *MoDELS’08*, LNCS 5301, pp. 782–796, 2008.

[17] P. Lahire, B. Morin, G. Vanwormhoudt *et al.*, “Introducing Variability into Aspect-Oriented Modeling Approaches,” in *MoDELS’07*, LNCS 4735, pp. 498–513, 2007.

[18] C. Ballagny, N. Hameurlain, and F. Barbier, “MOCAS: A State-Based Component Model for Self-Adaptation,” in *SASO’09*, pp. 206–215. IEEE, 2009.

[19] OMG, *A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems*, OMG, June 2008.