A Framework for Bridging the Gap Between Design and Runtime Debugging of Component-Based Applications

Guillaume Waignier, Sriplakich Prawee, Anne-Françoise Le Meur, Laurence Duchien

A Framework for Bridging the Gap Between Design and Runtime Debugging of Component-Based Applications*

Guillaume Waignier, Prawee Sriplakich, Anne-Françoise Le Meur, Laurence Duchien

Université Lille 1 - LIFL CNRS UMR 8022 - INRIA
59650 Villeneuve d'Ascq, France
{Guillaume.Waignier,Prawee.Sriplakich,
Anne-Francoise.Le_Meur,Laurence.Duchien}@inria.fr

Abstract. One concern when building application by assembling software components is to validate component interactions, *e.g.*, to ensure that components exchange compatible messages. This validation requires examining data values that are only known at runtime. In current practice, this validation is often performed manually at the code level, *i.e.*, architects need to insert validation code into the application code. This situation makes the interaction validation costly. Moreover, few platforms provide sufficient tools for supporting this validation.
As a solution, we propose CALICO, a model-based framework for runtime interaction validation. CALICO enables architects to specify validation concerns in the application model. It automatically propagates this specification to application code so that component interactions in the application can be checked at runtime. Based on the detected errors, CALICO allows architects to revisit the design to fix the detected errors, and then to repeat the runtime validation in an iterative process. This paper focuses on the integration of tools in CALICO for linking between validation specification at design time and validation realization at runtime. Moreover, we show how to extend CALICO to support multiple platforms with small development effort.

1 Introduction

CBSE is a widely used paradigm for creating complex software. It consists in building an application by assembling software component [1]. To ensure application reliability, CBSE software development usually includes checking the consistency of *component interactions*, which concerns the values of messages exchanged by components, and the order in which components exchange them. This validation aims to detect errors, such as an incompatibility of message values exchanged by components.

In a typical software process, architects perform this validation while iterating over the design, implementation and debugging tasks. The design task consists in elaborating application models describing a component assembly, which we refer to as *Architecture Description (AD) models*. This task also includes static validation of the AD models to ensure the consistency of component interactions [2]. The implementation task corresponds to generating and writing code for each component. The debugging task aims

* This work was partially funded by the French ANR TL FAROS project.

at checking whether the modeled application can execute correctly on a given platform. It consists in loading the components specified in the AD models on the target platform, and in detecting *runtime errors* on component interactions. This detection requires analysis of data whose values can only be known at runtime. To take into account runtime errors detected during the debugging task, architects reiterate over the design, implementation and debugging tasks, *i.e.*, they edit the AD models to fix the problems, reimplement the added or modified components and redebug the application.

These three tasks require using the combination of several software artifacts, such as models, code, configuration files, and involve multiple tools, such as model editor and model analysis tools for the design task, a code generator tool for the implementation task and a deployment tool for the debugging task. However, provided by different vendors, these tools are usually poorly integrated, making difficult to architects to use them in an iterative way. Indeed, the architects usually have to manually transfer data among tools and to convert data among the formats required by the different tools. This task is tedious and prone to errors.

Furthermore, few platforms provide a sufficient tool set for enabling architects to develop a reliable application. For example, platforms such as CCM [3] and Fractal [4] provide no support to statically check component interaction inconsistency at design-time. Thus, when developing an application on such a platform, architects suffer from the lack of tools for detecting inconsistencies in an early phase of the software development process. Moreover, few platforms provide mechanisms for debugging component interactions at runtime; consequently, architects can suffer from workload in coding the debugging support in their application code.

To enable architects to perform efficient iterative software development on a platform of their choice, we propose CALICO, the Component AssembLy Interaction Control framewOrk. CALICO provides architects with a set model-based tools for editing AD models and checking model consistency during the design task, for generating skeleton code during the implementation task and for validating component interactions at runtime during the debugging task. First, CALICO enables architects to address *validation concerns* at the model level, *i.e.*, it allows architects to specify, on the AD models, the component interactions they need to check. For example, architects can specify the assertion that the data a component receives must be of valid format. CALICO ensures that this verification is performed at runtime, *i.e.*, during the debugging task. When an error is detected, architects can use CALICO to fix the problems in the AD models. Each modification in the models is propagated to the running system by performing dynamic updates of the component assembly. Architects can reiterate over the design, implementation and debugging tasks until the system becomes stable.

This paper focuses on the set of tools that CALICO proposes for bridging the gap between the design and the runtime debugging. When designing CALICO tools, we aimed to provide genericity, *i.e.*, the tools manipulate AD concepts that are common to several platforms, and extensibility, *i.e.*, the tools can be extended with functionalities to validate applications running on different platforms. We explain how these tools are integrated to enable architects to perform iterative software process in a continuous way. We describe how CALICO can be extended to support multiple platforms, so that architects can gain the ability to develop reliable software even on platforms that cur-

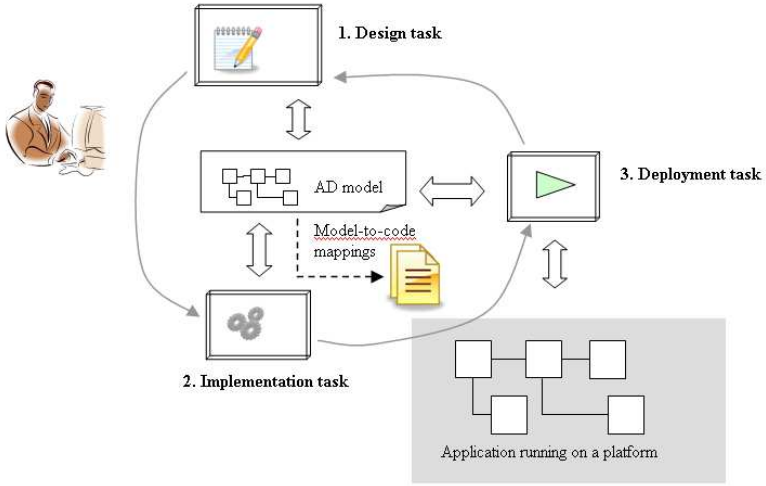

Fig. 1. Overview of CALICO

rently lack support for component interaction validation. Complementary to this paper, details on the AD model semantics and the model analysis techniques for validating component interactions are presented in our previous papers [5] and [6].

This paper is organized as follows. Sections 2 gives a rapid overview of CALICO and illustrates how it can be used in an iterative software development process. Sections 3 describes how CALICO handles the multi-platform support and the iterative software development process. Finally, Section 4 compares CALICO with other work and Section 5 concludes.

2 Iterative Software Development

CALICO is a framework for designing and debugging component-based applications on multiple platforms. It covers three tasks in an iterative software development process: design, implementation and debugging (c.f. Figure 1). CALICO integrates these tasks to enable continuity between them. This integration is based on using common application's AD models. The AD models are created during the design task, used to both generate code during the implementation task and finally load the application during the debugging task.

2.1 Overview

In the **design task**, CALICO enables the architect to design and analyze the specification of an application. The application is described with a set of AD models containing both the Platform Independent Model (PIM) part and Platform Specific Model (PSM) part. The PIM part includes the *system structure model*, which depicts the application architecture in terms of components and connections. It also contains three models to

express, respectively, the structural constraints, the control flow going in and out of components, and the constraints on the values of the exchanged messages. The PSM part refines the PIM part by adding platform-specific details.

During the design phase, CALICO offers a platform-independent approach to address validation concerns. First, to address static validation, it provides architects with a tool to statically verify model consistency regarding component interactions. Second, to address runtime validation, CALICO offers also the notion of *validation points*, which specify the interactions that needs to be checked at runtime, *i.e.*, the value and order of the messages exchanged between connected components.

In the **implementation task**, CALICO provides the generation of skeleton code, *i.e.*, component interfaces and component implementation classes, which are to be filled with business code. This code generation reduces manual errors that architects could have made in code writing. It guarantees that the component implementation respects the component specification expressed in the system structure AD model.

CALICO also addresses validation concerns by translating validation points into components, called *interceptors*. The interceptors are responsible for performing, at runtime, the checks described in the validation points. Furthermore, CALICO automatically instruments the whole application code to add the debugging support. The instrumented code reifies runtime information, *e.g.*, control flow of component interactions, that the interceptors require for performing the checks. This instrumentation is generic enough to allow the architect to develop a reliable system on any given platform, even if the platform lacks support for reifying the runtime information. From the architect's viewpoint, the instrumentation is driven by models, and the mechanisms to realized the instrumentation are transparent to the architect.

During the **debugging task**, CALICO loads the application onto the underlying platform and instantiates the needed interceptors. At runtime, the interceptors check the component interactions and report to the architects the detected errors. The error detection may lead architects to revisit the design task, *i.e.*, updating the AD models, for fixing the detected error. Then, when the architect iterates over the implementation and debugging tasks, CALICO dynamically propagates the changes into the running system, without reloading the whole system.

2.2 An Iterative Scenario: the PHR System

To illustrate the use of CALICO, we present an example software development scenario for the French Personal Health Record system (PHR) system [7]. The PHR system aims to enable heterogeneous clients, used by health-care staffs, to access Health Record documents stored in servers. Several clients are connected to the Web Portal service, which is in charge of dispatching the client requests to the corresponding Data Store servers. In this system, each client only supports particular document formats, *e.g.*, HTML and plain text, while the format of documents provided by heterogeneous servers may be different. Consequently, when an architect executes tests on the system, he/she notices the following errors: some clients receive documents in unsupported formats.

Consider now that the PHR system has been designed and implemented with CALICO (cf. Figure 2). To debug the PHR system, the architect needs to identify which clients, servers and documents are causing the errors. Accordingly, the architect can

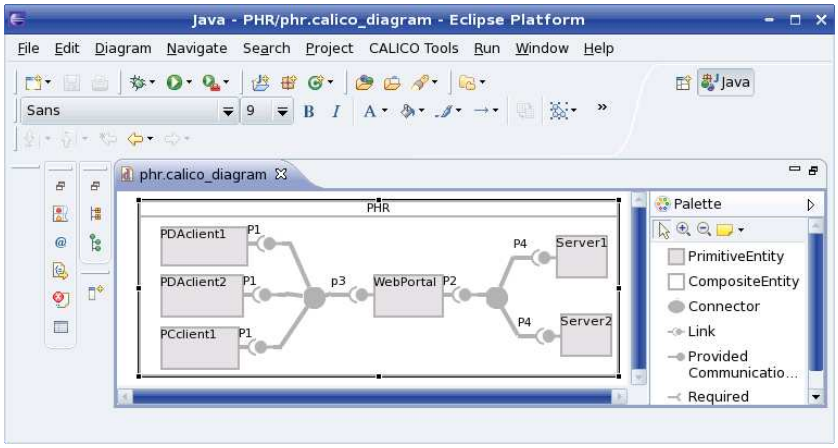

Fig. 2. PHR system

place validation points on the server ports, in order to detect errors as early as possible. The validation points express the checks to perform on the documents returned by the servers, to identify potential incompatibility with requesting clients. These checks rely on control flow information to determine the client involved in each control flow. CALICO propagates the validation points into the running system by generating and loading the interceptors. After the interceptor insertion, the architect can run tests again. The inserted interceptors enable the architect to obtain the source of the errors.

Having identified the error source, the architect is able to provide a solution to fix the problem. This solution consists in inserting Data Converter components between the Web Portal and incompatible clients to convert the data returned by the servers into a format compatible to the client, *e.g.*, converting Microsoft Word into plain text documents. This solution is realized during another software development iteration, *i.e.*, using CALICO to update the design and to propagate the changes to the running system.

3 The CALICO Architecture

We describe in this section two parts of CALICO: the model and tool parts. With respect to the model part, we present the system structure AD metamodel that is used by architects to define an application architecture, *i.e.*, AD model. Based on this metamodel, we define the Update metamodel representing the changes in the AD model performed by the architects at each software development iteration. The tool part concerns the framework architecture that allows the integration of tools involved in the design, implementation and debugging tasks, and enables the framework extension to support multiple platforms.

3.1 The Metamodels Part

The system structure AD metamodel offers metaclasses that represent the concepts of Component, Port and Connector, which are common concepts in component-based plat-

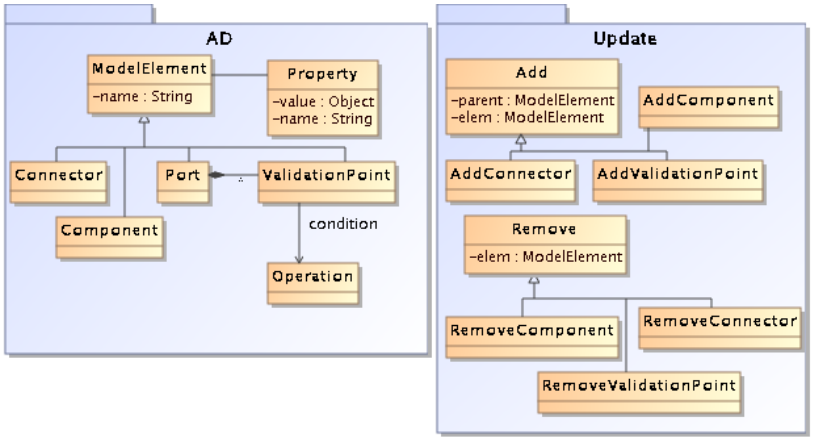

Fig. 3. AD metamodel and Update metamodel

forms (cf. Figure 3). These concepts have already been presented in [6], in this paper we focus on the concepts concerning the iterative software development support and the multi-platform support.

Linking the design to runtime debugging. The system structure AD metamodel enables architects to insert validation points on component ports, which specifies that a message sent or received through the port needs to be reified for debugging purpose. A validation point contains a message filtering condition. This condition is a boolean operation that is evaluated on the data values contained in the messages. In the PHR example, this operation checks if the documents sent by the server is compatible with the client. The condition specification greatly reduces the architect's workload when debugging, by selecting the reified information that the architect needs to analyze.

Supporting Multiple Platforms. Each platform usually requires extra information in the AD model in order to define platform-specific configuration, such as component implementation or connection type, *e.g.*, synchronous, asynchronous, etc. [8]. CALICO offers means to specify Platform Specific Model (PSM) properties and to validate them. The AD metamodels adopts the concept of *property* of Acme [9] for defining these PSM properties. A property is a name-value information entity that can be attached to a model element. The validation is encapsulated in a *Platform Profile*. A Platform Profile defines a set of PSM properties that an architect is allowed to specify, and the set of constraints, written in OCL, that must be satisfied by the application model in order to make sure that the application can be loaded on a given platform.

For example, the Platform Profile of CCM defines the property `impl-class` for specifying the implementation class of each component, and the property `type` for specifying whether the component port is a `facet`, `receptacle`, `event_sink` or `event_source`. This Platform Profile specifies also that only ports with compatible types can be connected.

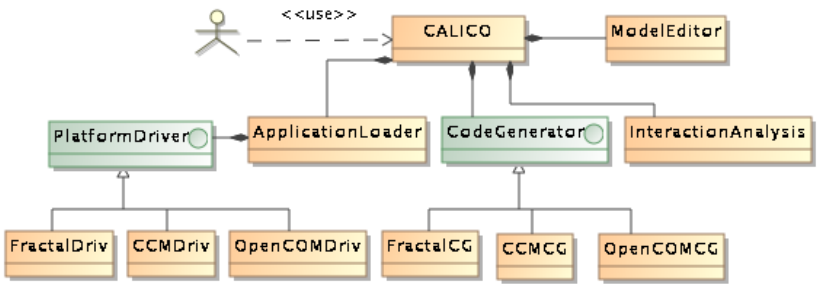

Fig. 4. Tool integration in CALICO

Supporting Iteration. Based on the AD metamodel, we define the Update metamodel, which represents the changes between two versions of an AD model (cf. Figure 3). The Update model contains the sequence of `Add`/`Remove` operations that define the addition/removal of model elements, *i.e.*, components, connectors and validation points, to/from the AD model. The `Add` operation specifies the model element to be added and the location to add it, *i.e.*, the parent model element. For example, the operation `AddValidationPoint` specifies the validation point to be added, and the component port where to attach the validation point.

The Update model is only used internally within CALICO, the architect does not need to have access or edit the instance of this metamodel. Rather this instance is generated automatically by comparing the two model versions: the version representing the running application and the version that has been modified by the architect.

3.2 Tool Part

CALICO offers a set of tools for supporting the design, implementation and debugging tasks (cf. Figure 4). During the design task, an architect uses the *Model Editor* tool to edit the AD model, and refine it with PSM properties. Then, the *Interaction Analysis* tool checks the constraints on the AD model to ensure consistency. During the implementation task, the architect uses the *Code Generator* tool both to generate the skeleton code of the components and to instrument the application code to support debugging. During the debugging task, the *Application Loader* tool updates the running application, accordingly to the AD model.

Linking the design to runtime debugging. To support debugging, CALICO offers translation of validation points into interceptors in the running application. An interceptor is a non-business component inserted between two ports of interacting business components. It provides a pair of client and server ports to forward the messages sent and received by the business components. Its role consists in evaluating if the method arguments of the port respects the condition specified by the validation point. We have chosen to realize the interception mechanisms using regular components for genericity purpose, *i.e.*, this approach can be realized on any platforms. Moreover, this approach enables us to dynamically add interception mechanisms to the running application.

Supporting Multiple Platforms. We propose the notion of plugins to integrate platform-specific functionalities into the framework. A plugin contains the following elements: a Platform Profile, a Code Generator and a Platform Driver.

The *Platform Profile* is an OCL file defining the PSM properties that are checked by the Interaction Analysis tool during the design task.

The *Code Generator* implements the Code Generator interface, defined in CALICO. It defines three main operations. The operation `genSkeleton` generates the skeleton code of the business components from the system structure AD model. The operation `genInterceptor` produces the implementation of the interceptors. Its implementation can be based on platform-specific code templates. The operation `instrument` is used for instrumenting the whole application code for inserting the debugging support, *e.g.*, reifying control flow. Its implementation is based on aspect weaving.

The *Platform Driver* implements the Platform Driver interface. Its role consists in masking the heterogeneity of platform APIs from Application Loader. It defines seven operations for initializing the platform, creating/removing components/connectors and pausing/resuming components. The Application Loader tool masks the notion of validation points from the Platform Driver by translating them into interceptor components. The Platform Driver does not need to make a distinction between the interceptor components and the business components. Thus, the developer of the driver only needs to provide mechanisms for realizing the `Add`/`Remove` operations of components/connectors. This approach enables the validation point translation mechanism to be refactored, so that it can be reused in multiple platforms.

In the design of the Platform Driver interface, we address the problem that platform APIs require heterogeneous parameters. As a solution, we make the AD model, which contains PSM properties, available to the Platform Driver. This approach enables architects to extend the AD model so that it contains sufficient platform-specific information for being loaded on an underlying platform. For example, in order to instantiate a component, the driver can access the PSM properties of the component definition for identifying the binary code of the component.

Supporting Iteration. To support iteration, CALICO enables an architect to update the running application through the system structure AD model. The Application Loader offers an operation for updating the running application, accordingly to the new AD model that the architect has edited. This update is performed as follows. First, the Application Loader generates the Update model by comparing the new AD model with the one it has preserved since last reconfiguration, *i.e.*, the AD model of the currently running application. The comparison consists in matching structural elements of both models. The elements that do not match are identified as added or removed elements. This comparison mechanism is applicable for both hierarchical-component models, *e.g.*, Fractal [4], and flat-component models, *e.g.*, OpenCCM [3], OpenCOM [10]. Furthermore, if the behavior of a component changes, this component is identified as removed and added, since in our approach we consider components as black boxes. Accordingly, the only way to change the component behavior is to change the component code and to reload the component in the system. In the second step, the Application Loader translates the operations "add/remove validation points" of the Update model into operations for adding/removing interceptor components. Finally, the Plat-

form Driver executes the operations in the Update model. To avoid that an update to a model puts the system into an inconsistent state, the Update model is first applied on a clone of the AD models, which is statically verified, before applying the update on the running system.

4 Related Work

Several script languages, such as FScript [11], are intended to support dynamic reconfiguration of component-based applications. In these approaches, architects define system update in terms of operations such as adding/ removing components. These approaches do not prevent architects from writing a script that creates system inconsistencies. In CALICO, the Update model, equivalent to a script, is automatically generated by comparing the model of the running system and the new model designed by architects. Our approach is result-oriented: it allows architects to check the preview of the update result. Furthermore it simplifies the architect's task in updating the application design, by eliminating the architect's need to learn the script language.

Plastik is an ADL/Component runtime integration meta-framework [12]. Like CALICO, it offers a platform-independent language to describe the software architecture, by using an extension of Acme/Armani [9], and mechanisms for loading the application on the underlying platform. However, to our knowledge, it is implemented only for the OpenCOM platform [10]. Moreover, there is a gap between the design, implementation and debugging tasks. First, architects must manually write the component code accordingly to the architecture description. Second, architects have to implement their components so that they are able to reify runtime information. These manual efforts can be error-prone. On the contrary, CALICO automates the transition between the design, development and deployment tasks, by automatically generating skeleton code and interceptor code.

5 Conclusion

This paper presents CALICO a generic and extensible framework for bridging the gap between the design, implementation and debugging tasks. Building such a framework requires dealing with different information involved in each of the tasks. The model task requires high-level specifications of component structures and behaviors, for enabling consistency checking; the implementation and debugging tasks require platform-specific, low-level specifications, which enable the executability of the modeled application. To tackle this challenge, we propose a tool integration approach based on generic, yet extensible, AD models. This approach reduces the workload of architects in realizing transitions between the design, implementation and debugging tasks: from the architect point of view, it may look like that the execution and debugging tasks are directly performed on the application model. Moreover, the multiple platform support of CALICO enables an analysis tool to be written once and for all, and each platform supported by CALICO can then benefit from this tool.

CALICO has been implemented and is fully integrated with Eclipse [1]. This allows architects to do all iterative development tasks without leaving the integrated environ-

[1] CALICO is available at http://calico.gforge.inria.fr

ment, *i.e.*, graphically designing the application model, checking model consistency, examining and correcting the model inconsistency, generating skeleton code, adding business code, compiling the code, and executing and debugging the application. The entire framework uses EMF to manipulate the models, which are the core data used by all tools.

We have successfully extended the framework with plugins for three platforms: OpenCCM [3], Fractal [4] and OpenCOM [10]. For each plugin, the implementation efforts consist in studying the platform-specific component model to define the Platform Profile, in developing the code templates and code instrumentation aspects to implement Code Generator, and, in implementing the Platform Driver, based on the Platform's API. As experience feedback, we have found out that parts of the code templates and aspects can be reused in several platforms. Moreover, by refactoring the Application Loader's mechanism that translates validation points to interceptors, we were able to reuse it in several platforms. Our experience shows that, extending a CALICO to support new platforms, OpenCCM or OpenCOM, can be done with small effort, *i.e.*, one man-week for each platform.

In near future, we plan to add the support for iterative development on service-oriented platforms, in particular those based on Web Services, such as SCA. We also consider implementing Platform Drivers that support complex component connectors, such as stream-based, secured and broadcasting connectors.

References

1. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. on Software Engineering **26**(1) (January 2000)
2. Hoare, C.: Communicating Sequential Processes. Prentice Hall (June 2004)
3. OMG: CORBA Component Model, v4.0, formal/06-04-01. (April 2006)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in Java. In: Proceedings of the 7th International Symposium Component-Based Software Engineering. Volume 3054 of LNCS., Springer (May 2004)
5. Waignier, G., Le Meur, A.F., Duchien, L.: Architectural specification and static analyses of contractual application properties. In: Proceedings of the 4th International Conference on the Quality of Software-Architectures (QoSA 2008). (2008) To appear.
6. Waignier, G., Sriplakich, P., Le Meur, A.F., Duchien, L.: A model-based framework for statically and dynamically checking component interactions. In: Proceedings of the ACM/IEEE 11th International Conference on MODELS 2008. (2008)
7. Nunziati, S.: Personal health record www.d-m-p.org/docs/EnglishVersionDMP.pdf.
8. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: ICSE. (2000) 178–187
9. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Foundations of Component-Based Systems. Cambridge University Press (2000)
10. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: Opencom v2: A component model for building systems software. In: IASTED Software Engineering and Applications. (2004)
11. David, P.C., Ledoux, T.: An aspect-oriented approach for developing self-adaptive fractal components. In: Software Composition. (2006) 82–97
12. Batista1, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Proceedings of the 2nd European Workshop (EWSA 2005). (2005)