Toward Structural and Behavioral Analysis For Component Models

Hanh-Missi TRAN
LIFL*
missi@lifl.fr

Philippe BEDU
EDF - R&D†
philippe.bedu@edf.fr

Laurence DUCHIEN
LIFL*
laurence.duchien@lifl.fr

Hai-Quan NGUYEN
EDF - R&D†
quan.nguyenhai@edf.fr

Jean PERRIN
EDF - R&D†
jean.perrin@edf.fr

ABSTRACT

Component use is becoming more and more prevalent every day. Indeed advantages such as greater productivity represent interesting qualities for the creation of industrial applications. Important efforts are made to help engineers through the improvement of the design and the description of components and through the specification of contracts. However most of the approaches that associate components and contracts propose only run-time checking. In software architecture design, it would be useful to consider contracts when we check the validity of the architecture. Our work takes place in the context of the RM-ODP(Reference Model for Open Distributed Processing) and more precisely the DASIBAO methodology. This paper presents a component-based model associated with several contracts and it describes some verifications that can be performed on them.

Keywords

RM-ODP, structural and behavioral analysis,component-based architecture, ADL, assembly

1. INTRODUCTION

Software architecture is used as a main part of the specification of component-based systems. Reasoning about software architectures improves design, program understanding, and formal analysis. Nowadays most of the software architects tend to agree that the design of sophisticated and software-intensive distributed applications has to be performed according to different viewpoints. As proposed in UML's "4+1" viewpoint model [9], IEEE1471 [3] or ISO RM-ODP (Reference Model for Open Distributed Processing)[2], the separation of concerns during architecture specification helps the designers to manage the complexity of the development process. Viewpoints give some guidance on the models to be produced during a design process as well as the objectives of these models. EDF R&D (Electricity of France Group) has opted for a methodology of architecture design based on RM-ODP, which recommends the separation of stakeholders concerns and proposes five viewpoints. On top of this reference model, EDF is implementing an incremental specification method called DASIBAO [8]. This method defines the different transformations between the viewpoints and particularly between the models carried by each viewpoint. This approach takes all its dimension within the framework of the OMG-MDA (Object Management group Model-Driven Architecture) [1] where designers are expected to produce collections of models from different viewpoints.

However the various models built with this specification method have to guarantee an acceptable level of quality for the system to be created. Our work focuses on the fourth viewpoint which specifies the abstract structure of a model and its deployment in a distributed environment. This work is quite original because it introduces formal analysis abilities in the global architecture specification process based on RM-ODP. This paper presents our approach to model architectures in ODP's systems and a set of tools integrated in the CASBA (Component Assembly Structural & Behavioral Analyzer) system that has been developed jointly by LIFL and EDF R&D. Section 2 introduces our composition model. Then, Section 3 presents some structural elements that can be checked such as the meta-model conformance, the operation signature compatibility and the pre- and post-conditions. Section 4 proposes some behavioral contracts for handling behavioral composition. We propose a description language to specify the behavioral contracts and we check some liveness and safety properties. To avoid an explosion of the number of states, we propose some cuts to reduce behavioral composition. Then we present the results of our verification tool applied to a real application used by EDF. Finally, we conclude and give some perspectives.

*Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
59655 - Villeneuve d'Ascq Cedex, France

†Electricité de France - Research Division
1, Avenue du General de Gaulle
92141 - Clamart Cedex, France

2. COMPONENT TYPES MODEL

Our architecture is specified with components. The concept of component used in this architecture is based on the following definition from Szyperski[14]: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" The elements needed to describe our component model and the relations between them are taken into account in the metamodel represented in Figure 1.

2.1 Components

Our approach only handles component as component type. Components may have attributes which represent their state. Moreover they are described by provided and required interfaces. A component communicates with other components through its interfaces. In our model, an interface is represented by a port which is associated with a single service. A service is specified by its signature which is composed of a name and of ingoing and outgoing parameters.

2.2 Components assembly

The model does not include explicit connectors between components. Nonetheless, if an architect needs one, he can model it in a component. Communications between components or more precisely between their ports are specified by an assembly link. The semantics of a link corresponds to a synchronous call from the required port to the provided port. The choice regarding a port structure involves that a required port can only be bound to a provided port whereas a provided port can be bound to several required ports.

2.3 Components composition

In order to build a complex architecture, we use **composite components**. They are differentiated from **primitive components** because they contain subcomponents which may be primitive components and also composite components hence a recursive definition of a component. The ports of a composite component are called **delegated ports**. Indeed a call to a provided port of a composite component is forwarded to a provided port of one of its subcomponents. Moreover a call from a required port of a composite component results from the forwarding of a call from a required port from one of its subcomponents.

2.4 Functional contracts

The conditions of validity of a component assembly are improved by associating an **assembly contract** composed of a **pre-condition** and a **post-condition** to each port. These conditions focus on the attributes of the component and on the parameters of the signature. Thus in addition to the verification of the signature compatibility between two linked ports, there is an analysis that checks respectively the compatibility of the precondition and postcondition of a port with the precondition and postcondition of the linked port.

Furthermore **behavioral contracts** are added to the components. These contracts describe the expected behavior of a component and are used to generate the behavior of the components assembly. An appropriate tool has been developed to check some properties on it.

3. STRUCTURAL VERIFICATION

Our tool provides basic verification features common to several ADLs (Architecture Description Languages). It offers a syntactic verification by checking if the components model is in accordance with the metamodel. The metamodel is translated into an XML schema to use the mechanism of validation of XML documents against XML schema.

Another analysis focuses on the assembly links. The previous mechanism verifies that a required port is bound to only one provided port. Moreover there is an analysis on the compatibility of the signatures of bound ports that is based only on their parameters. Indeed we consider that the name of the signature can only be used to identify the port and that it does not give the semantics of the service of the port. The compatibility of a port signature with another uses the notions of covariance and contravariance. These concepts are bound to the paradigm of object-oriented programming. They are distinct mechanisms: *"The so-called contravariance rule correctly captures the subtyping relation. A covariant rule, instead, characterizes the specialization of code"*[6]. The compatibility of port signatures is characterized by three levels. There is a **strong compatibility** of the signature of the required port with the signature of the provided port when there is a contravariance of the ingoing parameters and a covariance of the outgoing parameters. If there is a covariance instead of a contravariance or vice versa, there is only a **weak compatibility**. There is **no compatibility** when there is neither a covariance nor a contravariance between the parameters.

The compatibility of the assembly contracts associated with a required port and a provided port is checked on top of these verifications. Three levels characterize this compatibility. In our model, pre-conditions and post-conditions specify conjunctions and disjunctions of linear inequations. Given *P1* and *P2* two logical formulas and x_1,...,x_n the values in these logical formulas, the strong compatibility can be checked by:
$\forall (x_1,...,x_n) \in \{(x_1,...,x_n)/\mathrm{P1}(x_1,...,x_n)=\mathrm{true}\}, \mathrm{P1}(x_1,...,x_n) \Rightarrow \mathrm{P2}(x_1,...,x_n)$

Our tool uses CiaoProlog[4], an implementation of Prolog that offers a constraint solver on real values. Because CiaoProlog finds values that solve the constraints, it checks in fact:
$\neg (\exists (x_1,...,x_n) \in \{(x_1,...,x_n)/\mathrm{P1}(x_1,...,x_n)=\mathrm{true}\}, \neg(\mathrm{P1}(x_1,...,x_n) \Rightarrow \mathrm{P2}(x_1,...,x_n)))$

Given *P1* and *P2* two logical formulas and x_1,...,x_n the values in these logical formulas, the weak compatibility can be checked by:
$\exists (x_1,...,x_n) \in \{(x_1,...,x_n)/\mathrm{P1}(x_1,...,x_n)=\mathrm{true}\}, \mathrm{P1}(x_1,...,x_n) \cap \mathrm{P2}(x_1,...,x_n) \neq \emptyset$

Our pre-conditions and post-conditions are written in a language very close to Java Modeling Language (JML)[10] which can be used as a design by contract language for Java. For example, instead of using the keyword *result*, the post-condition is specified with the name of the outgoing parameter. More complex pre-conditions and post-conditions could be expressed by the use of boolean expressions on top of the arithmetical ones. The research of solutions could be made by associating the use of a constraint solver and a SAT solver.

The previous verifications correspond to a structural approach. The analysis of the behavioral contracts performs verifications in a dynamic approach.

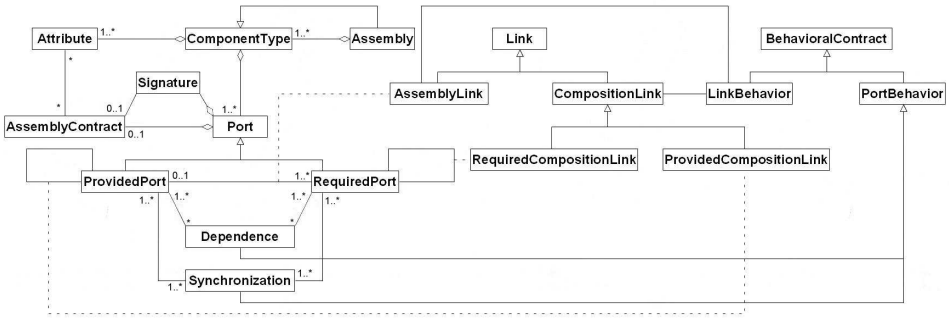

Figure 1: Metamodel of components composition

4. BEHAVIORAL VERIFICATION

In order to check if the system runs as required, the behavior of the component assembly is analyzed. Several parts of the behavior of the component assembly need to be described: they are called **behavioral contracts**. This section first presents the language to specify the behavioral contracts and then the verifications that are performed.

4.1 Component behavior

A component can be viewed as either a black box or a white box. Thus its external behavior can be distinguished from its internal one. The external and internal behaviors are the same in the case of a primitive component. Its behavior is composed of the ways its ports are called. These communications are described in behavioral contracts. We distinguish **dependences** from **synchronizations** as shown in the metamodel (figure 1). A dependence represents a behavioral contract which specifies the internal communications of a component. It consists of the specification of the required ports that are needed by a provided port and the way they are called by the provided port. A synchronization deals with the concurrency issues.

In order to get the internal behavior of a composite component, we add the behavior specified by the composition links, the assembly links between its subcomponents to the behavior of its subcomponents. A communication through either an assembly link or a composition link represents a call from a port to another port.

4.2 Description language

The execution of a service is represented by a sequence of events. Given a sequence S, its execution is translated into S.call $\rightarrow$ S.begin $\rightarrow$ S.end $\rightarrow$ S.return. This means that S is called, then begins, ends and finally returns a value. The operator $\rightarrow$ symbolizes a partial order because the relation is not reflexive but symmetric and transitive. The operators of the description language are based on this operator.

The sequence and alternative operators are both used in the specification of dependences and synchronizations. Let A et B be two services. $A;B$ means that A is executed and then B is executed. It is translated into $A.call \rightarrow A.begin \rightarrow A.end \rightarrow A.return \rightarrow B.call \rightarrow B.begin \rightarrow B.end \rightarrow B.return$. $A|B$ means that either A or B is executed. Thus possible traces are either $A.call \rightarrow A.begin \rightarrow A.end \rightarrow A.return$ or $B.call \rightarrow B.begin \rightarrow B.end \rightarrow B.return$.

The other two operators are only used in dependences. The call operator represents the communication from a provided port to the required ports and the parallel operator represents parallel composition of services. Let A, B and C be three services. $A\{B\}$ means that the execution of A is composed of the execution of B. It is translated into $A.call \rightarrow A.begin \rightarrow B.call \rightarrow B.begin \rightarrow B.end \rightarrow B.return \rightarrow A.end \rightarrow A.return$. $A\{B||C\}$ means that the execution of A is composed of the interleaving execution of B and C.

The last operator * is used to specify that there can be an undetermined number of executions of a service or that this service is not executed. For example, if we specify $A^*|B$, it means that either the service A is executed several times or not at all or the service B is executed. This operation may be used to describe loops.

The null sequence symbolized by $\emptyset$ comes in addition to these operators. It indicates that no service is executed.

4.3 Behavior verification

In order to analyze on the behavior of a component model, we transform the behavioral elements into FSP (Finite State Process)[11] processes. We operate this translation first by generating a behavior formed of the behaviors of the composition and assembly links and the dependences and then by making a composition of it with the synchronizations. The analysis uses a verification tool for concurrent systems, named LTSA (Labelled Transition System Analyser)[12], which supports FSP and a LTL (Linear Temporal Logic) checker to check safety and liveness properties such as deadlocks or absence of reachability.

This approach works well when applied on small architectures. However large architectures are represented by complex hierarchical component structure and the analysis of the behavior of such architectures may lead to state explosion problems. Thus the behavior of composite components has to be minimized. This approach is close to the TRACTA approach[7]. Both are based on FSP but because the behavioral contracts are described with our own language, the produced FSP specifications do not use all the features of this language.

4.4 Behavior minimization

The first way to obtain the external behavior from the internal behavior of a composite component is to use the FSP minimization operator. However the FSP processes describe a composite component internal behavior. Moreover each time an analysis uses its external behavior, the internal behavior of each subcomponent is minimized again. To address this problem, we have decided to perform this minimization with our description language. This operation

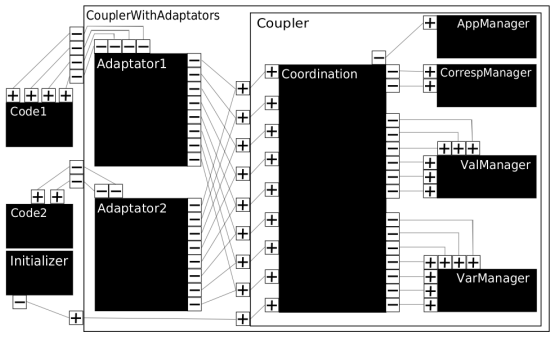

Figure 2: Architecture of the CALCIUM coupler

aims at producing the behavior of a composite component as if it were a primitive component. Thus this behavior is composed of dependences and synchronizations.

The transitivity of the operator $\rightarrow$ is the basis of the reduction of an internal behavior into an external one. Indeed the calling operator is based on the operator $\rightarrow$ and the behaviors of the dependences and the assembly and delegation links use the calling operator. The beginning of the minimization consists in transforming the ports that do not call any other ports into the null sequence. Then the transitivity of the calling operator allows the behavior to be reduced.

The minimization of synchronizations may lead to the loss of information on the behavior. Because our language is based on services and not on events, it is currently not tractable enough to realize a minimization on it. Our verification tool uses the minimization based on our behavior language but only minor changes would be needed in order to use the minimization feature in FSP.

5. RESULTS

The most significant example verified by our verification tool is an existing application from EDF. The figure 2 gives an idea of the complexity of the architecture. Required and provided ports are symbolized respectively by - and +. The example represents the use of a generic coupler of scientific code named CALCIUM[5] which first version was developped in 1994. This coupler is used to study the interactions between codes of different domains in physics. It manages the exchange of values between the codes.

Several assembly and behavioral contracts are added to the architecture shown in the figure 2. The structural verification takes some time to be performed, due to the number of compatibilities of assembly contracts to be checked. The behavioral analysis can not be done because of the explosion of the number of states. For example, the potential state space for the behavior of the component *Coordination* is wide of 2^{170} states and an usual desktop computer does not have enough memory to handle it.

6. CONCLUSION AND FUTURE WORK

Our component model is used to specify functional architectures in the computational viewpoint. We integrate contracts into the model to carry out strong verifications on the components model. Assembly contracts add conditions to the validity of components assembly. Moreover behavioral contracts specify the communications within a component. Furthermore the use of behavior minimization associates hierarchical composition with behavioral composition in our architecture. The verifications we describe are implemented by tools in CASBA. These tools can be called from a graphical interface integrated in the modelling tool ArgoUML [13] which allows the design and the analysis of component model based on our metamodel.

The structural and behavioral verifications of our component model represent only a part in our approach of architecture building. Indeed we now need to specify how to go from a computational viewpoint, which is the fourth viewpoint in RM-ODP, to an engineering viewpoint which is its last one. Thus our goal is to integrate non functional requirements in functional architectures in order to produce technical architectures. This leads us to propose a new concept called **architectural figure** inspired by previous works on the reuse of architectural systems such as the architectural patterns and styles. This architectural figure represents a component model slightly different from our previous model which allows us to transform the functional architecture into a technical one. Thus our future work will focus on providing tools to describe figures associated with quality attributes, to realize the transformation of functional architectures into technical ones with architectural figures and to analyse quality aspects of the produced architectures.

7. REFERENCES

[1] www.omg.org/mda.

[2] Iso/iec, open distributed processing reference model - parts 1, 2, 3, 4. ISO 10746 or ITU-T X.901, 1995.

[3] Recommended practice for architectural description. IEEE Standard P1471, 2000.

[4] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. Lopez, and G. Puebla. The ciao prolog system: A next generation logic programming environment. Technical Report 3/97.1, CLIP, April 2004.

[5] C. Caremoli and J.-Y. Berthou. *CALCIUM V2: Guide d'utilisation.*

[6] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[7] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures.* PhD thesis, Imperial College of Science Technology and Medecine, University of London, March 1999.

[8] A. W. Group. Dasibao: Methodology for architecturing odp systems. Technical report, EDF R&D, 2002.

[9] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(5):42–50, November 1995.

[10] G. Leavens and Y. Cheon. Design by contract with jml. Draft paper, March 2004.

[11] J. Maggee and J. Kramer. *Concurrency - State Models and Java Program.* John Wiley & Sons, 1999.

[12] J. Maggee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[13] J. E. Robbins. *Cognitive Support Features for Software Development Tools.* PhD thesis, University of California, Irvine, 1999.

[14] C. Szyperski. *Component Software - Beyond Object Programming.* 1998.