Component Model with Support of Mobile Architectures

Marek Rychlý

Department of Information Systems,
Faculty of Information Technology, Brno University of Technology,
Božetěchova 2, 612 66 Brno, Czech Republic,
rychly@fit.vutbr.cz

Abstract. Common features of current information systems have significant impact on software architectures of the systems. The systems can not be realised as monoliths, formal specification of behaviour and interfaces of the systems' parts are necessary, as well as specification of their interaction. Moreover, the systems have to deal with many problems including the ability to clone components and to move the copies across a network (component mobility), creation, destruction and updating of components and connections during the systems' runtime (dynamic reconfiguration), maintaining components' compatibility, etc. In this paper, we present the component model with support of mobile architectures and outline its formal basis. We also review the related research on the current theory and practice of formal component-based development of software systems.

Keywords: Software Architecture, Component-Based Development, Component Model, Formal Specification

1 Introduction

Increasing globalisation of information society and its progression create needs for extensive and reliable information technology solutions. Common requirements for current information systems include adaptability to variable structure of organisation, support of distributed activities, integration of well-established (third party) software products, connection to a variable set of external systems, etc. Those features have significant impact on software architectures of the systems. The systems can not be realised as monoliths, exact specification of functions and interfaces of the systems' parts are necessary, as well as specification of their communication and deployment. Therefore, the information systems of organisations are realised as networks of quite autonomous, but cooperative, units communicating asynchronously via messages of appropriate format. Unfortunately, design and implementation of the systems have to deal with many problems including the ability to clone components and to move the copies across a network (i.e. *component mobility*), creation, destruction and updating of components and connections during the systems' runtime (i.e. *dynamic reconfiguration*), maintaining components' compatibility, etc. [4]

Moreover, those distributed information systems are getting involved. Their architectures are evolving during runtime and formal specifications are necessary, particularly in critical applications. Design of the systems with *dynamic architectures* (i.e. architectures with dynamic reconfigurations) and *mobile architectures* (i.e. dynamic architectures with component mobility) can not be done by means of conventional software design methods. In most cases, these methods are able to describe semi-formally only sequential processing or simple concurrent processing bounded to one component without advanced features such as dynamic reconfiguration.

The *component-based development* (CBD, see [10]) is a software development methodology, which is strongly oriented to composability and re-usability in a software system's architecture. In the CBD, from a structural point of view, a software system is composed of *components*, which are self contained entities accessible through well-defined *interfaces*. Connection of compatible interfaces of cooperating components is realised via *connectors*. Actual organisation of components interconnected via connectors is called *configuration*. *Component models* are specific meta-models of software architectures supporting the CBD, which define syntax, semantics and composition of components.

Although the CBD can be the right way to cope with the problems of the distributed information systems, it has some limitations in formal description, which restrict the full support for the mobile architectures. Those restrictions can be delimited by usage of formal bases that do not consider dynamic reconfigurations and component mobility, strict isolation of control and business logic of components that does not allow full integration of dynamic reconfigurations into the components, etc.

In this paper, we propose the component model with support of mobile architectures and review the related current formal approaches to the CBD. The remainder of this paper is organised as follows. In Section 2, we introduce our proposed approach in more detail. In Section 3, we review the main approaches that are relevant to our subject. In Section 4, we discuss advantages and disadvantages compared with the reviewed approaches. To conclude, in Section 5, we summarise our approach, current results and briefly outline the future work.

2 Component Model

In this section, we describe our approach to the component model with support of mobile architectures. The Figure 1 describes an outline of the component model's meta-model. Three basic entities represent the core entities of a component based architecture: a component, an interface and a connector.

The *component* is an active communicating entity in a component based software system. In our approach, the component consists of component abstraction and component implementation. The *component abstraction* (`CompAbstraction` in the meta-model) represents the component's specification and behaviour given by the component's formal description (semantics of services provided by the component). The *component implementation* (`CompImplementation`) represents

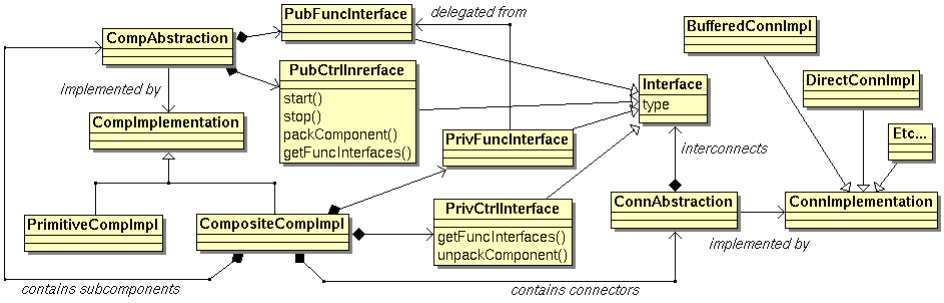

Fig. 1. Diagram of the meta-model (an outline).

specific implementation of the component's behaviour (implementation of the services). The implementation can be primitive or composite. The *primitive implementation* (`PrimitiveCompImpl`) is realised directly, beyond the scope of architecture description (it is "a black-box"). The *composite implementation* (`CompositeCompImpl`) is decomposable on a system of subcomponents at the lower level of architecture description (it is "a grey-box"). The subcomponents are represented by component abstractions (`CompAbstraction`).

The *interface* of a component (descendants of entity `Interface`) can be sorted according to its relative location compared with the component into public and private interfaces. The *public interfaces* (`PubFuncInterface` and `PubCtrlInterface`) are required or provided (attribute `type` of the entity) by a component to its neighbouring components at the same level of the architecture description (i.e. not to subcomponents of a neighbouring component, for example). The *private interfaces* (`PrivFuncInterface` and `PrivCtrlInterface`), which exist only in composite components, are the components' public interfaces delegated into the components' composite implementation where they are available for the components' subcomponents. According to functionality of interfaces, we distinguish functional interfaces and control interfaces. The *functional interfaces* (`PubFuncInterface` and `PrivFuncInterface`) represent business oriented services. The *control interfaces* (`PubCtrlInterface` and `PrivCtrlInterface`) provide services for components' introspection (e.g. `getFuncInterfaces()`) and changes of an architecture and behaviour (`start()` and `stop()`). The services for changes of an architecture are, for example, `packComponent()` and `unpackComponent()` for a component's transformation into and from a transmittable message, respectively.

The *connector* is responsible for a reliable communication between required and provided interfaces. It consists of connector abstraction and connector implementation. The *connector abstraction* (`ConnAbstraction`) represents an abstract connection of a pair of compatible interfaces. The *connector implementation* (`ConnImplementation`) represents specific implementation of the connector, which depends e.g. on communication style (buffered and unbuffered connection) or a type of synchronisation (blocking and output non-blocking).

2.1 Behaviour and Support of Mobile Architectures

We focus on behaviour particularly related to *the features of mobile architectures*, i.e. on creation and destruction of components and connections and on passing of components. Evolution of an architecture begins in the state where initialisation of the architecture is finished. Then, *a new component* can be created by means of duplication of an existing component[1] where the new component can be placed as a subcomponent of a parent component or sent via its outgoing connections (via provided interfaces). *Destruction of a component* can be done automatically after releasing of its provided interfaces (the component is forgotten when there are no outgoing connections). *Creation of new connections* between two interfaces can be done also by means of passing of components. A component, which is creating a new connection, receives a component with a target interface and obtains the interface via the component's control interface. This enables a connection to interconnect subcomponents of two different parent components if one of the subcomponent is accessible via passing of components, i.e. to share one subcomponent between many parent components. *Destruction of a connection* can be done directly via connected interface (actually, the connection is forgotten, so no destruction is needed). *The passing of a component* is realised by means of its control interface (the component is "packed" into a message) and control interface of target component (the message is "unpacked" and the component will become a subcomponent of the target component).

As it follows from the description of behaviour, the connections can interconnect required functional interfaces with (provided[2]) control interfaces. This allows to build systems where functional (business) requirements imply changes of the systems' architectures.

2.2 Formal Description

The component model's formal description can be realised by means of the process algebra π-calculus, known as the *calculus of mobile processes* [6], which allows modelling of systems with dynamic communication structures (i.e. mobile processes). The description is based on our previous research on distributed information systems as systems of asynchronous concurrent processes [8] and the mobile architecture's features in such systems [9].

Formal description of a C component's behaviour can be expressed as the π-calculus process $C = (C_f|C_c) + stop.start.C$. It is a non-deterministic choice between a parallel composition of processes, which represent its functional part C_f and control part C_c, and the process that waits for "start" after it receives "stop" via names *start* and *stop*, respectively. Interfaces of the component C are represented by free names in the process C, by its parameters, if the C is denoted as a parametric process $C(p_1, \ldots, p_m)$.

For component abstraction C, the definition of the process C_f is given by a designer of a system or a component, which contains the component C as its

[1] by means of `packComponent()` and `unpackComponent()`

[2] in our approach, the control interfaces are only provided (they provide a control)

subcomponent. It represents required functional behaviour of the system's or component's part. For component primitive implementation C, the definition of the process C_f is given by description of functional behaviour of the C component's implementation according to its specification (the implementation is "a blackbox"). For component composite implementation, the definition of the process C_f is a parallel composition of n processes $C_1(p_{1,1}, \ldots, p_{1,m_1})$, $\ldots$, $C_n(p_{n,1}, \ldots, p_{n,m_n})$ that represent component abstractions of the C component's subcomponents, and their interconnections. For each connection between provided interfaces represented by names $p_1, \ldots, p_u$ and required interfaces represented by names $q_1, \ldots, q_v$ of the subcomponents, we can define parametric π-calculus process for binding the interfaces

$$B(p_1, \ldots, p_u, q_1, \ldots, q_v) = \sum_{i=1}^{u} \sum_{j=1}^{v} q_j(x).\overline{p_i}\langle x\rangle.B(p_1, \ldots, p_u, q_1, \ldots, q_v) \qquad (1)$$

As it has been described in the previous section, the most of the features of mobile architectures can be reduced to the component mobility feature. In the π-calculus, this feature can be described as passing of π-calculus processes: directly by means of higher order π-calculus or indirectly by means of passing of names ("an executor example" in [6]). The indirect method in terms of systems of asynchronous concurrent processes has been described in [9]. We can define processes that realises pack and unpack control interfaces. Let $C(p_1, \ldots, p_m)$ be a π-calculus process representing behaviour of a component C with m interfaces. Then, we define parametric process P_C to send (export) all interfaces $p_1, \ldots, p_m$ of the component C via name p, and process U_C to receive (import) the interfaces via name p into a context of another process, as follows

$$\begin{aligned} P_C(p, p_1, \ldots, p_m) &= p(x)(\overline{x}\langle p_1\rangle \ldots \overline{x}\langle p_n\rangle) \\ U_C(p, p_1, \ldots, p_m) &= (x)(\overline{p}\langle x\rangle.x(p_1) \ldots x(p_n)) \end{aligned} \qquad (2)$$

Those processes implement interfaces `pack()` and `unpack()`, respectively. The control part of the component C is defined as $C_c = !P_C$. The process U_C is used by a component, which is a destination of passing of the C's interfaces.

3 Related Work

There have been proposed several component models [5]. In this section, we focus on two contemporary component models supporting some features of dynamic architectures and formal descriptions.

The **component model Fractal** [2] is a general component composition framework with support for dynamic architectures. A Fractal component is formed out of two parts: a controller and a content. The content of a composite component is composed of a finite number of nested components. Those subcomponents are controlled by the controller ("a membrane") of the enclosing component. A component can be shared as a subcomponent by several distinct components. A component with empty content is called a primitive component.

Every component can interact with its environment via operations at external interfaces of the component's controller, while internal interfaces are accessible only from the component's subcomponents. The interfaces can be of two sorts: client (required) and server (provided). Besides, a functional interface requires or provides functionalities of a component, while a control interface is a server interface with operations for introspection of the component and to control its configuration. There are two types of directed connections between compatible interfaces of components: primitive bindings between a pair of components and composite bindings, which can interconnect several components via a connector.

Behaviour of Fractal components can be formally described by means of *parametrised networks of communicating automata* language [1]. Behaviour of each primitive component is modelled as a finite state *parametrised labelled transition system* (pLTS) – a labelled transition system with parametrised actions, a set of parameters of the system and variables for each state. Behaviour of a composed Fractal component is defined using a *parametrised synchronisation network* (pNet). It is a pLTS computed as a product of subcomponents' pLTSs and a transducer. The transducer is a pLTS, which synchronises actions of the corresponding LTSs of the subcomponents. When synchronisation of the actions occurs, the transducer changes its state, which means reconfiguration of the component's architecture. Also behaviour of a Fractal component's controller can be formally described by means of pLTS/pNet. The result is composition of pLTSs for binding and unbinding of each of the component's functional interfaces (one pLTS per one interface) and pLTS for starting and stopping the component.

In the **component model SOFA** [7], a part of *SOFA project (SOFtware Appliances)*, a software system is described as a hierarchical composition of primitive and composite components. A component is an instance of a template, which is described by its frame and architecture. The frame is a "black-box" specification view of the component defining its provided and required interfaces. Primitive components are directly implemented by described software system – they have a primitive architecture. The architecture of a composed component is a "grey-box" implementation view, which defines first level of nesting in the component. It describes direct subcomponents and their interconnections via interfaces. The connections of the interfaces can be realised via connectors, implicitly for simple connections or explicitly. Explicit connectors are described in a similar way as the components, by a frame and architecture. The connector frame is a set of roles, i.e. interfaces, which are compatible with interfaces of components. The connector architecture can be simple (for primitive connectors), i.e. directly implemented by described software system, or compound (for composite connectors), which contains instances of other connectors and components.

The SOFA uses a *component definition language* (CDL) for specification of components and *behaviour protocols* (BPs) for formal description of their behaviours. The BPs are regular-like expressions on the alphabet of event tokens representing emitting and accepting method calls. Behaviour of a component (its interface, frame and architecture) can be described by a BP (interface, frame and architecture protocol, respectively) as the set of all traces of event tokens

generated by the BP. The architecture protocols can be generated automatically from architecture description by a *CDL compiler*. A *protocol conformance relation* ensures the architecture protocol generates only traces allowed by the frame protocol. From dynamic architectures, the SOFA allows only a dynamic update of components during a system's runtime. The update consists in change of implementation (i.e. an architecture) of the component by a new one. Compatibility of the implementations is guaranteed by the conformance relation of a protocol of the new architecture and the component's frame protocol.

Recently, the SOFA team is working on a new version of the component model. The **component model SOFA 2.0** [3] aims at removing several limitations of the original version of SOFA – mainly the lack of support of dynamic reconfigurations of an architecture, well-structured and extensible control parts of components, and multiple communication styles among components.

4 Discussion

The component model proposed in this paper is able to handle mobile architectures, unlike the SOFA that supports only a subset of dynamic architectures or the Fractal/Fractive, which does not support components mobility. As is described in Section 2.2, the π-calculus provides fitting formalism.

Moreover, the proposed semantics permits to combine provided control and required functional interfaces, e.g. in comparison with the Fractal/Fractive that also provides control and functional interfaces. Regardless, in some cases, it separation description and verification of control and functional parts is needed. The possible solution can be an application of typed π-calculus, which distinguishes a type of names, and replacing of some communication patterns that use control-functional bindings by special π-calculus constructions (e.g. a special stop/start processes recursively controlling also subcomponents of a component, which is stopped/started). Regardless, for mobile architectures, the ability to combine control and functional interfaces is necessary.

The next feature of the component model is partially independence of a component's specification from its implementation. This feature is similar to the SOFA's component-template relationship. It allows to control behaviour of a primary component's implementation, define a composite component's border that isolates its subcomponents, which is called "a membrane" in the Fractal, etc. Our goal is to expand the specification-implementation relationship of components so it allows runtime replacements of the components' implementations without need to stop a component during replacement, in comparison with the SOFA. We believe it can be achieved by means of component mobility.

However, all above mentioned features will be parts of an ongoing work.

5 Conclusion and Future Work

This paper describes the component model with support of mobile architectures. The component model splits a distributed information system into primitive and

composite components according of decomposability of the system's parts, the components' functional and control interfaces according to types of required or provided services, and connectors realising a communication layer between the components. The components and the connectors are described at different levels, as their specifications and implementations. Semantics of the entities can be described by means of π-calculus processes.

An ongoing work is related to completing exact description of the component model's formal semantics. Future work is mainly related to realisation of a supporting environment, which allows integration of the model into a software development process, including integration of verification tools and implementation support.

This research has been supported by the Grant Agency of Czech Republic grants No. 102/05/0723 "A Framework for Formal Specifications and Prototyping of Information System's Network Applications" and by the Research Plan No. MSM 0021630528 "Security-Oriented Research in Information Technology".

References

1. T. Barros. *Formal specification and verification of distributed component systems.* PhD thesis, Université de Nice – INRIA Sophia Antipolis, Nov. 2005.
2. E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium, Feb. 2004.
3. T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006*, pages 40–48, Seattle, USA, Aug. 2006. IEEE Computer Society.
4. J. Král and M. Žemlička. Autonomous components. In *SOFSEM 2000: Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*, pages 375–383. Springer, 2000.
5. K.-K. Lau and Z. Wang. A survey of software component models (second edition). Pre-print CSPP-38, School of Computer Science, The University of Manchester, Manchester M13 9PL, UK, May 2006.
6. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:41–77, Sept. 1992.
7. F. Plášil, D. Bílek, and R. Janeček. SOFA/DCUP: Architecture for component trading and dynamic updating. In *4th International Conference on Configurable Distributed Systems*, pages 43–51, Los Alamitos, CA, USA, May 1998. IEEE Computer Society.
8. M. Rychlý. Towards verification of systems of asynchronous concurrent processes. In *Proceedings of 9th International Conference Information Systems Implementation and Modelling (ISIM'06)*, pages 123–130. MARQ, Apr. 2006.
9. M. Rychlý and J. Zendulka. Distributed information system as a system of asynchronous concurrent processes. In *MEMICS 2006 Second Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 206–213. Faculty of Information Technology BUT, Oct. 2006.
10. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley Professional, second edition, Nov. 2002.