e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

A Component Model with Support of Mobile Architectures and Formal Description

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 these 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' run-time (dynamic reconfiguration), maintaining components' compatibility, etc. In this paper, we present the component model that addresses component mobility including dynamic reconfiguration, allows to combine control and functional interfaces, and separates a component's specification from its implementation. We focus on the formal basis of the component model in detail. We also review the related research on the current theory and practice of formal component-based development of software systems.

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 organisations, 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 [7]. Unfortunately, design and implementation of those 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' run-time (i.e. *dynamic reconfiguration*), maintaining components' compatibility, etc. [6]

Moreover, distributed information systems are getting involved. Their architectures are evolving during a run-time 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 com-

ponent without advanced features such as dynamic reconfiguration.

The *component-based development* (CBD, see [17]) 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*. A connection of compatible interfaces of cooperating components is realised via their *bindings* (connectors). Actual organisation of interconnected components 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.

This paper proposes a high-level component model addressing the mentioned issues. The model allows dynamic reconfigurations and component mobility, defined combination of control and business logic of components, and separation of a component's specification from its implementation. The paper also introduces a formal basis for description of the component model's semantics, i.e. the structure and behaviour of the components.

The remainder of this paper is organised as follows. In Section 2, we introduce the component model in more detail. In Section 3, we provide the formal basis for description of the component model. In Section 5, we review main approaches that are relevant to our subject. In Section 6, we discuss advantages and disadvantages of our component model and its formal description compared with the reviewed approaches and outline the future work. To conclude, in Section 7, we summarise our approach and current results.

2. Component Model

In this section, we describe our approach to the component model. The component model is presented in two views: structural and behavioural. At first, in Section 2.1, we introduce the component model's meta-model, which describes basic entities of the component model and their relations and properties. The second view, in Section 2.2, is focused on behaviour of the component model's entities, especially on the component mobility.

2.1. Meta-model

The Figure 1 describes an outline of the component model's meta-model[1] in the UML notation [20]. Three basic entities represent the core entities of a component based architecture: a component, an interface and a binding (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 a specific implementation of the component's behaviour (an implementation of the services). The implementation can be primitive or composite. The *primitive implementation* (`CompImplPrimitive`) is realised directly, beyond the scope of architecture description (it is "a black-box"). The

[1] The figured diagram can not describe additional constraints, e.g. a composite component "contains" bindings that interconnect only interfaces of the component's subcomponents, not interfaces of its neighbouring components, etc.

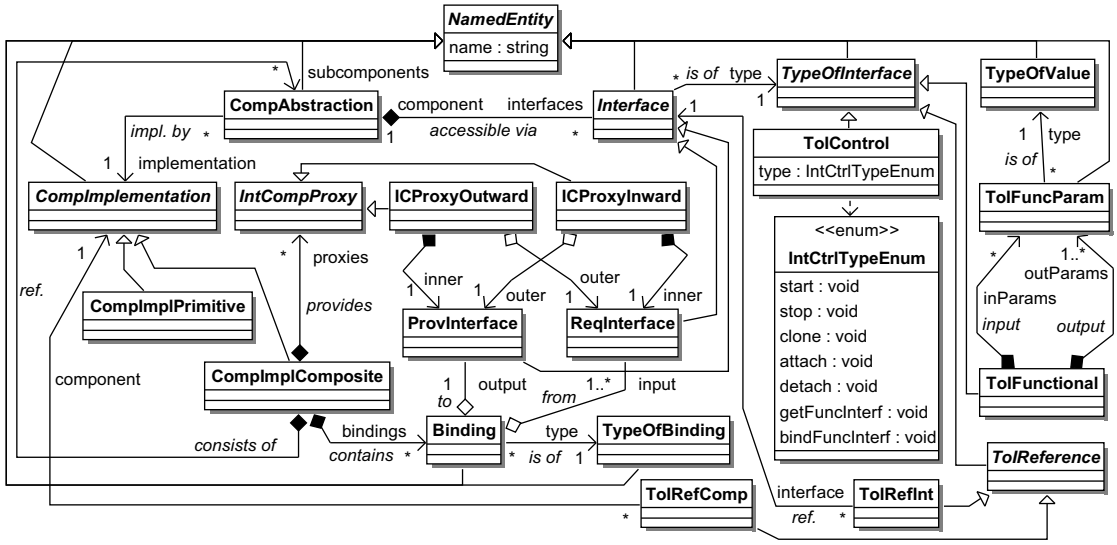

Figure 1. The meta-model of the component model (the UML notation [20])

composite implementation (`CompImplComposite`) is decomposable on a system of subcomponents at the lower level of architecture description (it is "a grey-box"). Those subcomponents are represented by component abstractions (`CompAbstraction` and relation "consists of").

Interfaces of a component are described in relation to the component's abstraction (relation "accessible via" from `CompAbstraction`). We distinguish two types of *interfaces*: required and provided (`ReqInterface` and `ProvInterface`, respectively), according to the type of services required or provided by the component from or to its neighbouring components, respectively, at the same level of hierarchy of components (i.e. not from or to subcomponents of a neighbouring component, for example). Moreover, the composite components' implementations (`CompImplComposite`) provide special *internal interfaces*, which are available only for the component's subcomponents and make accessible the component's external interfaces (i.e. the interfaces described in relation to `CompAbstraction`). The entity `ICProxyInward` connects a composite component's external provided interface to the component's internal required interface, while the entity `ICProxyOutward` connects a composite component's internal provided interface to the component's external required interface (the relations "outer" and "inner" and vice versa).

According to the functionality of interfaces, we can distinguish functional, control and reference interfaces (described by `TypeOfInterface`). The *functional interfaces* (`ToIFunct-ional`) represent business oriented services with typed input and output parameters (`ToIFuncParam` and `TypeOfValue`). The *control interfaces* (`ToIControl` and its attribute's `type`) provide services for obtaining references to a component's provided functional interfaces (type `getFuncInterfaces`), for binding a component's required functional interfaces (type `bind-FuncInterface`), and for changes of behaviour (types `start` and `stop`) and architecture. The services for changes of architecture are `clone`, `attach` and `detach` for obtaining references to a fresh copy of a component (type "cloning"), attaching of a new component as a subcomponent and detaching of an old subcomponent, respectively. The *reference interfaces* (`ToIReference`) are able to transmit references to components or interfaces, which is required to support component mobility.

Finally, the *binding* describes connection of required and provided interfaces of the identical types and of components at the same level of the hierarchy into a reliable communication link (entity `Binding`). The type of a binding (`TypeOfBinding`) can specify a communication

style (buffered and unbuffered connection), a type of synchronisation (blocking and output non-blocking), etc.

2.2. Behaviour and Support of Mobile Architectures

The previous section introduces the structure of the component model. A system described by means of the component model is one component with provided and required interfaces, which represent the system's input and output actions, respectively. The component can be implemented as a primitive component or as a composite component. The primitive component is realised directly, beyond the scope of architecture description, while the composite component is decomposable at the lower level of hierarchy into a system of subcomponents communicating via their interfaces and their bindings.

Behaviour of a primitive component has to be defined by a developer, simultaneously with definitions of the component's interfaces. The primitive component is defined as "a black-box", i.e. its behaviour can be described as a dependence relation of input and output actions. Behaviour of a composite component depends on behaviour of its subcomponents, but it includes also a description of communication between connected interfaces of those subcomponents and processing of specific control actions in the component (e.g. requests for starting or stopping of the component and their distribution to the component's subcomponents, etc.).

In the following description, we focus on the behaviour of control parts of components 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 a system's architecture begins in the state where its initialisation is finished.

A new component can be created as a copy of an existing component by means of its control interface `clone`. The resulting new component is deactivated (i.e. stopped) and packed into a message, which can be sent via outgoing connections into different location (via interfaces of type `ToIRefComp`) where it can be placed as a subcomponent of a parent component (by means of `attach` interface), connected to local neighbouring components (by means of `bindFuncInterf` and `getFuncInterf` interfaces) and activated (by means of `start` interface). *Destruction of an old component* can be done automatically after deactivating of the component (by means of `stop` interface), releasing of all its provided interfaces and disconnecting from its parent component (by means of `detach` interface).

Creation of new connections between two compatible functional interfaces can be done by means of passing of functional interfaces (via interfaces of type `ToIRefInt`). At first, a reference to provided functional interface (a target interface) is obtained from a component (via control interface `getFuncInterf`). This reference is sent via outgoing connections into different location (via interfaces of type `ToIRefInt`), but only in the same parent component and at the same level of hierarchy of components (i.e. crossing the boundary of a composite component is not allowed). The reference is received by a component with compatible required functional interface (a source interface) and a binding of this interface to referenced interface is created (by means of control interface `bindFuncInterf`). *Destruction of a connection* can be done by rebinding of a required interface participating in this connection.

As it follows from the description of behaviour, the connections can interconnect only interfaces of the same types. Moreover, dynamic creation of new connections and destruction of existing connection are permitted only for functional interfaces (type `ToIFunctional`). Those *restrictions*, together with the restriction of passing of interfaces' references described in the previous paragraph, prevent *architectural erosion* and *architectural drift* [11], which are caused by uncontrollable evolution of dynamic and mobile architecture resulting into degradation of the components' dependencies over time. In the component model, the architecture of control interfaces and their interconnections, which allow evolution and component mobility, is a static architecture.

Despite those restrictions, combining of actions of functional interfaces with actions of control interfaces is permitted inside primitive components. This allows to build systems where functional (business) requirements imply changes of a systems' architectures.

3. Formal Description

In this section, formal description of behaviour of the component model's entities is presented. The Section 3.1 provides an introduction to the process algebra π-calculus, which is used in description in Section 3.2. The description is based on our previous research on distributed information systems as systems of asynchronous concurrent processes [13] and the mobile architecture's features in such systems [15, 14].

3.1. The π-Calculus

The process algebra π-*calculus*, known also as a *calculus of mobile processes* [10], is an extension of Robin Milner's *calculus of communicating systems* (CCS). This section briefly summarises the fundamentals of the π-calculus, a theory of mobile processes, according to [16]. The following theoretical background is required for the component model's formal description in Section 3.2. The π-calculus allows modelling of systems with dynamic communication structures (i.e. mobile processes) by means of two concepts:

a process – an active communicating entity in a system, primitive or expressed in π-calculus (denoted by uppercase letters in expressions)[2],

a name – anything else, e.g. a communication link (a port), variable, constant (data), etc. (denoted by lowercase letters in expressions)[3].

Processes use names (as communication links) to interact, and pass names (as variables, constants, and communication links) to another process by mentioning them in interactions. The names received by a process can be used and mentioned by it in further interactions (as communication links). This "passing of names" permits mobility of communication links.

Processes evolve by performing actions. The capabilities for action are expressed via three kinds of prefixes ("output", "input" and "unobservable", as it is described later). We can define the π-calculus processes, their subclass and the prefixes as follows.

Definition 1 (π-calculus). *The* processes*, the* summations*, and the* prefixes *of the π-calculus are given respectively by*

$$\begin{aligned} P &::= M \mid P \mid P' \mid (z)P \mid !P \\ M &::= 0 \mid \pi.P \mid M + M' \\ \pi &::= \overline{x}\langle y\rangle \mid x(z) \mid \tau \end{aligned}$$

We give a brief, informal account of semantics of π-calculus processes. At first, process 0 is a π-calculus process that can do nothing, it is the *null process* or *inaction*. If processes P and P' are π-calculus processes, then following expressions are also π-calculus processes with formal syntax according to the Definition 1 and given informal semantics:

- $\overline{x}\langle y\rangle.P$ is an *output prefix* that can send name y via name x (i.e. via the communication link x) and continue[4] as process P,
- $x(z).P$ is an *input prefix* that can receive any name via name x and continue as process P with the received name substituted for every free occurrence[5] of name z in the process,
- $\tau.P$ is an *unobservable prefix* that can evolve invisibly to process P, it can do an internal (silent) action and continue as process P,
- $P + P'$ is a *sum* of capabilities of P together with capabilities of P' processes, it proceeds as either process P or process P', i.e. when a sum exercises one of its capabilities, the others are rendered void,
- $P \mid P'$ is a *composition* of processes P and P', which can proceed independently and can interact via shared names,

[2] A parametric process is also called "an agent".
[3] The names can be called according to their meanings (e.g. a port/link, a message, etc.).
[4] The prefix ensures that process P can not proceed until a capability of the prefix has been exercised.
[5] See the Definition 2.

- $(z)P$ is a *restriction* of the scope[6] of name z in process P,
- $!P$ is a *replication* that means an infinite composition of processes P or, equivalently, a process satisfying the equation $!P = P \mid !P$.

The π-calculus has two name-binding operators. The binding is defined as follows.

Definition 2 (Binding). *In each of $x(z).P$ and $(z)P$, the displayed occurrence of z is* binding *with* scope P. *An occurrence of a name in a process is* bound *if it is, or it lies within the scope of, a binding occurrence of the name, otherwise the occurrence is* free.

In our notations, we will omit a transmitted name, the second parts of input and output prefixes in a π-calculus expression, if it is not used anywhere else in its scope (e.g. instead of $(x)((y)\overline{x}\langle y\rangle.0 \mid x(z).0)$, we can write $(x)(\overline{x}.0 \mid x.0)$).

Since the sum and composition operators are associative and commutative (according to the relation of structural congruence [10]) they can be used with multiple arguments, independently of their order. Also an order of application of the restriction operator is insignificant. We will use the following notations:

- for $m \geq 3$, let $\prod_{i=1}^{m} P_i = P_1 \mid P_2 \mid \ldots \mid P_m$ be a *multi-composition* of processes $P_1, \ldots, P_m$, which can proceed independently and can interact via shared names,
- for $n \geq 2$ and $\tilde{x} = (x_1, \ldots, x_n)$, let $(x_1)(x_2)\ldots(x_n)P = (x_1, x_2, \ldots, x_n)P = (\tilde{x})P$ be a *multi-restriction* of the scope of names $x_1, \ldots, x_n$ to process P.

We will omit the null process if the meaning of the expression is unambiguous according to the above-mentioned equations (e.g. instead of $\overline{x}\langle y\rangle.0 \mid x(z).0$, we can write $\overline{x}\langle y\rangle \mid x(z)$). Moreover, the following equations are true for the *null process*:

$$M + 0 = M \qquad P \mid 0 = P \qquad (x)0 = 0$$

The π-calculus processes can be parametrised. A parametrised process, an abstraction, is an expression of the form $(x).P$. We may also regard abstractions as components of input-prefixed processes, viewing $a(x).P$ as an abstraction located at name a. In $(x).P$ as in $a(x).P$, the displayed occurrence of x is binding with scope P.

Definition 3 (Abstraction). *An* abstraction *of* arity $n \geq 0$ *is an expression of the form $(x_1, \ldots, x_n).P$, where the x_i are distinct. For $n = 1$, the abstraction is a monoadic abstraction, otherwise it is a polyadic abstraction.*

When an abstraction $(x).P$ is applied to an argument y it yields process $P\{y/x\}$. Application is the destructor of abstractions. We can define two types of application: pseudo-application and constant application. The pseudo-application is defined as follows.

Definition 4 (Pseudo-application). *If $F \stackrel{def}{=} (\tilde{x}).P$ is of arity n and $\tilde{y}$ is length n, then $P\{\tilde{y}/\tilde{x}\}$ is an* instance *of F. We abbreviate $P\{\tilde{y}/\tilde{x}\}$ to $F\langle\tilde{y}\rangle$. We refer to this instance operation as* pseudo-application *of an abstraction.*

In contract to the pseudo-application that is only abbreviation of a substitution, the constant application is a real syntactic construct. It allows to describe a recursively defined process.

Definition 5 (Constant application). *A* recursive definition *of a process constant K is an expression of the form $K \triangleq (\tilde{x}).P$, where $\tilde{x}$ contains all names that have a free occurrence in P. A* constant application, *sometimes referred as an* instance *of the process constant K, is a form of process $K\lfloor\tilde{a}\rfloor$.*

Communication between processes (a computation step) is formally defined as a *reduction relation* $\rightarrow$. It is the least relation closed under a set of reduction rules.

Definition 6 (Reduction). *The* reduction relation, $\rightarrow$, *is defined by the following rules:*

$$\text{R-Inter}\ \frac{}{(\overline{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \rightarrow P_1 \mid P_2\{y/z\}}$$

$$\text{R-Tau}\ \frac{}{\tau.P + M \rightarrow P}$$

$$\text{R-Par}\ \frac{P_1 \rightarrow P_1'}{P_1 \mid P_2 \rightarrow P_1' \mid P_2}$$

$$\text{R-Res}\ \frac{P \rightarrow P'}{(z)P \rightarrow (z)P'}$$

$$\text{R-Struct}\ \frac{P_1 = P_2 \rightarrow P_2' = P_1'}{P_1 \rightarrow P_1'}$$

$$\text{R-Const}\ \frac{}{K\lfloor\tilde{a}\rfloor \rightarrow P\{\tilde{a}/\tilde{x}\}}\ K \triangleq (\tilde{x}).P$$

[6] The scope of a restriction may change as a result of interaction between processes.

The communication is described by the main reduction rule R-Inter. It means that a composition of a process proceeding as either process M_1 or the process, which sends name y via name x and continues as process P_1, and a process proceeding as either process M_2 or the process, which receives name z via name x and continues as process P_2, can perform a reduction step. After this reduction, the process is $P_1 \mid P_2\{y/z\}$ (all free occurrences of z in P_2 are replaced by y).

3.2. Description of the Component Model

A *software system* can be described by means of the component model as one component with provided and required interfaces, which represent the system's input and output actions, respectively. According to the component model's definition, every component can be implemented as a primitive component or as a composite component. Since a *primitive component* is realised as "a black-box", its behaviour has to be defined by its developer. This behaviour can be formally described as a π-calculus process, which uses names representing the component's interfaces, but also implements specific control actions provided by the component (e.g. requests to start or stop the component). On the contrary, a *composite component* is decomposable at the lower level of hierarchy into a system of subcomponents communicating via their interfaces and their bindings (the component is "a grey-box"). Formal description of the composite component's behaviour is a π-calculus process, which is composition of processes representing behaviour of the component's subcomponents, processes implementing communication between interconnected interfaces of the subcomponents and internal interfaces of the component and processes realising specific control actions (e.g. the requests to start or stop the composite component, but including their distribution to the component's subcomponents, etc.).

Before we define π-calculus processes implementing the behaviour of a component's individual parts, we need to define the *component's interfaces* within the terms of the π-calculus, i.e. as names used by the processes. The following names can be used in external or internal view of a component, i.e. for the component's neighbours or the composite component's subcomponents, respectively.

- external: $s_0, s_1, c, r_1^s, \ldots, r_n^s, p_1^g, \ldots, p_m^g$ (of a primitive or composite component),
- internal: $a, r_1'^s, \ldots, r_m'^s, p_1'^g, \ldots, p_n'^g$ (of a composite component only),

where n is a number of the component's required functional interfaces, m is a number of the component's provided functional interfaces (both from the external view) and the names have the following semantics:

via s_0 – a running component accepts a request for its stopping, which a composite component distributes also to all its subcomponents,

via s_1 – a stopped component accepts a request for its starting, which a composite component distributes also to all its subcomponents,

via c – a component accepts a request for its cloning and returns a new stopped instance of the component as a reply,

via r_i^s – a component accepts a request for binding given provided functional interface (included in the request) to the required functional interface r_i,

via p_j^g – a component accepts a request for referencing to the provided functional interface p_j that is returned as a reply,

via a – a composite component accepts a request for attaching its new subcomponent, i.e. for attaching the subcomponent's s_0 and s_1 names (stop and start interfaces), which can be called when the composite component will be stopped or started, respectively, and as a reply, it returns a name accepting the request to detach the subcomponent.

We should remark that there is a relationship between the names representing functional interfaces in the external view and the names representing corresponding functional interfaces in the internal view of the composite component. The composite component connects its external functional interfaces $r_1, \ldots, r_n$ (required) and $p_1, \ldots, p_m$ (provided) accessible via names

$r^s_1, \ldots, r^s_n$ and $p^g_1, \ldots, p^g_m$, respectively, to internal functional interfaces $p'_1, \ldots, p'_n$ (provided) and $r'_1, \ldots, r'_m$ (required) accessible via names $p'^g_1, \ldots, p'^g_n$ and $r'^s_1, \ldots, r'^s_m$, respectively. Requests received via external functional provided interface p_j are forwarded to the interface, which is bound to internal functional required interface r'_j (and analogously for interfaces p'_i and r_i).

3.2.1. Interface's References and Binding

At first, we define an auxiliary process $Wire$[7], which can receive a message via name x (i.e. input) and send it to name y (i.e. output) repeatedly till the process receives a message via name d (i.e. disable processing).

$$Wire \triangleq (x,y,d).(x(m).\overline{y}\langle m\rangle.Wire\lfloor x,y,d\rfloor \ + \ d)$$

Binding of a component's functional interfaces is done via control interfaces. These control interfaces provide references to a component's functional provided interfaces and allow to bind a component's functional required interfaces to referenced fictional provided interfaces of another local components. Process $Ctrl_{Ifs}$ implementing the control interfaces can be defined as follows

$$\begin{aligned}
SetIf &\triangleq (r,s,d).s(p).(\overline{d}.Wire\lfloor r,p,d\rfloor \mid SetIf\lfloor r,s,d\rfloor)\\
GetIf &\stackrel{def}{=} (p,g).g(r).\overline{r}\langle p\rangle\\
Plug &\stackrel{def}{=} (d).d\\
Ctrl_{Ifs} &\stackrel{def}{=} (r_1,\ldots,r_n,r^s_1,\ldots,r^s_n,p_1,\ldots,p_m,p^g_1,\ldots,p^g_m)\\
&\quad .(\prod_{i=1}^{n}(r^d_i)(Plug\langle r^d_i\rangle \mid SetIf\lfloor r_i,r^s_i,r^d_i\rfloor) \mid \prod_{j=1}^{m}!GetIf\langle p_j,p^g_j\rangle)
\end{aligned}$$

where names $r_1, \ldots, r_n$, $r^s_1, \ldots, r^s_n$, $p_1, \ldots, p_m$, $p^g_1, \ldots, p^g_m$ have been defined at the beginning of Section 3.2. Let us assume $Ctrl_{Ifs}$ shares its names $r_1, \ldots, r_n$ and $p_1, \ldots, p_m$ with a process implementing a component's core functionality via its required and provided interfaces, respectively. Pseudo-application $GetIf\langle p_j, p^g_j\rangle$ enables process $Ctrl_{Ifs}$ to receive a name x via p^g_j and to send p_j via name x as a reply (it provides a reference to an interface represented by p_j). Constant application $SetIf\lfloor r_i, r^s_i, r^d_i\rfloor$ enables process $Ctrl_{Ifs}$ to receive a name x via r^s_i, which will be connected to r_i by means of a new instance of process $Wire$ (it binds a required interface represented by r_i to a provided interface represented by x). To remove a former connection of r_i, a request is sent via r^d_i (in case it is the first connection of r_i, i.e. there is no former connection, the request is accepted by pseudo-application $Plug\langle r^d_i\rangle$).

In a composite component, the names representing external functional interfaces $r_1, \ldots, r_n$, $p_1, \ldots, p_m$ are connected to the names representing internal functional interfaces $p'_1, \ldots, p'_n$, $r'_1, \ldots, r'_m$. Requests received via external functional provided interface p_j are forwarded to the interface, which is bound to internal functional required interface r'_j (and analogously for interfaces p'_i and r_i). This is described in process $Ctrl_{EI}$.

$$\begin{aligned}
Ctrl_{EI} &\stackrel{def}{=} (r_1,\ldots,r_n,p_1,\ldots,p_m,r'_1,\ldots,r'_m,p'_1,\ldots,p'_n)\\
&\quad .\prod_{i=1}^{n}(d)Wire\lfloor r_i,p'_i,d\rfloor \mid \prod_{j=1}^{m}(d)Wire\lfloor r'_j,p_j,d\rfloor
\end{aligned}$$

[7] The process will be used also in the following parts of Section 3.2.

3.2.2. Control of a Component's Life-cycle

Control of a composite component's life-cycle[8] can be described as process $Ctrl_{SS}$.

$$
\begin{aligned}
Dist &\triangleq (p,m,r).(\overline{p}\langle m\rangle.Dist\lfloor p,m,r\rfloor \;+\; \overline{r})\\
Life &\triangleq (s_x,s_y,p_x,p_y).s_x(m).(r)(Dist\lfloor p_x,m,r\rfloor \mid r.Life\lfloor s_y,s_x,p_y,p_x\rfloor)\\
Attach &\stackrel{def}{=} (a,p_0,p_1).a(c_0,c_1,c_d)(d)\\
&\quad (c_d(m).\overline{d}\langle m\rangle.\overline{d}\langle m\rangle \mid Wire\lfloor p_0,c_0,d\rfloor \mid Wire\lfloor p_1,c_1,d\rfloor)\\
Ctrl_{SS} &\stackrel{def}{=} (s_0,s_1,a).(p_0,p_1)(Life\lfloor s_1,s_0,p_1,p_0\rfloor \mid !Attach\langle a,p_0,p_1\rangle)
\end{aligned}
$$

where names s_0 and s_1 represent the component's interfaces that accept stop and start requests, respectively, and name a that can be used to attach a new subcomponent's stop and start interfaces (at one step).

The requests for stopping and starting the component are distributed to its subcomponents via names p_0 and p_1. Constant application $Life\lfloor s_1,s_0,p_1,p_0\rfloor$ enables process $Ctrl_{SS}$ to receive a message m via s_0 or s_1. Message m is distributed to the subcomponents by means of constant application $Dist\lfloor p_x,m,r\rfloor$ via shared name p_x, which can be p_0 in case the component is running or p_1 in case the component is stopped. When all subcomponents accepted message m, it is announced via name r and the component is running or stopped and ready to receive a new request to stop or start, respectively.

Pseudo-application $Attach\langle a,p_0,p_1\rangle$ enables process $Ctrl_{SS}$ to receive a message via a, a request to attach a new subcomponent's stop and start interfaces represented by names c_0 and c_1, respectively. The names are connected to p_0 and p_1 via new instances of processes $Wire$. Third name received via a, c_d, can be used later to detach the subcomponent's previously attached stop and start interfaces.

3.2.3. Cloning of Components and Updating of Subcomponents

Cloning of a component allows to transport the component's fresh copy into different location, i.e. its subsequent attaching as a subcomponent of other component. The processes of the cloning can be described as follows

$$
\begin{aligned}
Ctrl_{clone} \triangleq\; & (x).x(k).(s_0,s_1,c,r_1^s,\ldots,r_n^s,p_1^g,\ldots,p_m^g,r,p)\\
& (\overline{k}\langle s_0,s_1,c,r,p\rangle \mid \overline{r}\langle r_1^s,\ldots,r_n^s\rangle \mid \overline{p}\langle p_1^g,\ldots,p_m^g\rangle\\
& \mid Component\langle s_0,s_1,c,r_1^s,\ldots,r_n^s,p_1^g,\ldots,p_m^g\rangle \mid Ctrl_{clone}\lfloor x\rfloor)
\end{aligned}
$$

where pseudo-application $Component\langle s_0,s_1,c,\, r_1^s,\ldots,r_n^s,p_1^g,\ldots,p_m^g\rangle$ with well-defined parameters describes behaviour of the cloned component. When process $Ctrl_{clone}$ receives a request k via name x, it sends names s_0,s_1,c,r,p via name k as a reply. The first three names represent "stop", "start" and "clone" interfaces of a fresh copy of the component. The process is also ready to send names representing functional requested and provided interfaces of the new component, i.e. names $r_1^s,\ldots,r_n^s$ via name r names $p_1^g,\ldots,p_m^g$ via name p, respectively, and to receive a new request.

The fresh copy of a component can be used to replace a compatible subcomponent of a composite component. The process of update, which describes the replacing of an old subcomponent with a new one, is not mandatory part of the composite component's behaviour and its implementation

[8] A primitive component handles stop and start interfaces directly.

depends on particular configuration of the component (e.g. if the component allows updating of its subcomponents, a context of the replaced subcomponent, which parts of the component have to be stopped during the updating, etc.). As an illustrative case, we can describe process $Update$ as follows

$$\begin{aligned} Update \triangleq\ & (u, a, s_0, s_d, r^s_1, \ldots, r^s_m, p^g_1, \ldots, p^g_n)(k, s'_d) \\ & .(\overline{u}\langle k\rangle.k(s'_0, s'_1, c, r', p').\overline{s_0}.\overline{a}\langle s'_0, s'_1, s'_d\rangle.\overline{s_d} \\ & .r'(r'^s_1, \ldots, r'^s_n).(x)(\overline{p^g_1}\langle x\rangle.x(p).\overline{r'^s_1}\langle p\rangle \ldots \overline{p^g_n}\langle x\rangle.x(p).\overline{r'^s_n}\langle p\rangle) \\ & .p'(p'^g_1, \ldots, p'^g_m).(x)(\overline{p'^g_1}\langle x\rangle.x(p).\overline{r^s_1}\langle p\rangle \ldots \overline{p'^g_n}\langle x\rangle.x(p).\overline{r^s_m}\langle p\rangle) \\ & .\overline{s'_1}.Update\lfloor u, a, s'_0, s'_d, r^s_1, \ldots, r^s_m, p^g_1, \ldots, p^g_n\rfloor) \end{aligned}$$

Process $Update$ sends via name u a request for a fresh copy of a cloned component. As a return value, it receives a vector of names representing all functional interfaces in a process describing behaviour of the new component, which will replace an old subcomponent in its parent component implementing the update process. Name a provides the parent component's internal control interface to attach the new subcomponent's stop and start interfaces (the s'_0 and s'_1 names) and an interface later used to detach the subcomponent (name s'_d). Name s_0 is used to stop the replaced subcomponent and name s_d is needed to detach the old subcomponent's stop and start interfaces. Finally, names $r^s_1, \ldots, r^s_m$, $p^g_1, \ldots, p^g_n$ represent a context of the updated subcomponent, i.e. connected interfaces of neighbouring subcomponents.

3.2.4. Primitive and Composite Components

In conclusion, we can describe the complete behaviour of primitive and composite components. Let's assume that process abstraction $Comp_{impl}$ with parameters $s_0, s_1,\ r_1, \ldots, r_n,\ p_1, \ldots, p_m$ describes behaviour of the core of a primitive component (i.e. excluding processing of control actions), as it is defined by the component's developer. Further, let's assume that process abstraction $Comp_{subcomps}$ with parameters $a,\ r'^s_1, \ldots, r'^s_m,\ p'^g_1, \ldots, p'^g_n$ describes behaviour of a system of subcomponents interconnected by means of their interfaces into a composite component (see Section 3.2.1). Names $s_0, s_1,\ r_1, \ldots, r_n,\ p_1, \ldots, p_m$ and names $a,\ r^s_1, \ldots, r^s_m,\ p^g_1, \ldots, p^g_n$ are defined at the beginning of Section 3.2.

Processes $Comp_{prim}$ and $Comp_{comp}$ representing behaviour of the mentioned primitive and composite components can be described as follows

$$\begin{aligned} Comp_{prim} \stackrel{def}{=}\ & (s_0, s_1, c, r^s_1, \ldots, r^s_n, p^g_1, \ldots, p^g_m)(r_1, \ldots, r_n, p_1, \ldots, p_m) \\ & .(Ctrl_{Ifs}\langle r_1, \ldots, r_n, r^s_1, \ldots, r^s_n, p_1, \ldots, p_m, p^g_1, \ldots, p^g_m\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Comp_{impl}\langle s_0, s_1, r_1, \ldots, r_n, p_1, \ldots, p_m\rangle) \end{aligned}$$

$$\begin{aligned} Comp_{comp} \stackrel{def}{=}\ & (s_0, s_1, c, r^s_1, \ldots, r^s_n, p^g_1, \ldots, p^g_m) \\ & .(a, r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n) \\ & (Ctrl_{Ifs}\langle r_1, \ldots, r_n, r^s_1, \ldots, r^s_n, p_1, \ldots, p_m, p^g_1, \ldots, p^g_m\rangle \\ & \mid Ctrl_{Ifs}\langle r'_1, \ldots, r'_m, r'^s_1, \ldots, r'^s_m, p'_1, \ldots, p'_n, p'^g_1, \ldots, p'^g_n\rangle \\ & \mid Ctrl_{EI}\langle r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Ctrl_{SS}\langle s_0, s_1, a\rangle \mid Comp_{subcomps}\langle a, r'^s_1, \ldots, r'^s_m, p'^g_1, \ldots, p'^g_n\rangle) \end{aligned}$$

where processes $Ctrl_{Ifs}$ represent behaviour of control parts of components related to their interfaces (see Section 3.2.1), processes $Ctrl_{clone}$ describe behaviour of a control part of components related to cloning of these components (see Section 3.2.3), process $Ctrl_{SS}$ represents behaviour of

a component's control part handling its stop and start requests (see Section 3.2.2), and process $Ctrl_{EI}$ describes behaviour of communication between internal and external functional interfaces of a component (see Section 3.2.1).

4. An Example

As an example, we describe a component based system for user authentication and access control. At first the system receives an input from an user in form $(username, password)$ and verifies the user's password in order to check the user's identity. If the user's password passes the verification, the system creates a new session handle reserved for the user. The session handle is connected to the system's core. It enables the user to access the system's core functionality and performs the access control according to the user's authorisation. Finally, the session handle is passed back to the user as a return value of the whole process.

The system is composed of

- Login component verifying the user's authentication and initiating the new session,
- Core component providing the system's core functionality,
- and Session component enabling the user to access the Core component according to the user's authorisation.

For simplicity, let's assume that component Session has only one input interface for the user's calls of the system's core without any explicit authorisation checks and component Core implements simple shared memory – one storage for all users with two interfaces: for saving and loading a value to and from the memory, respectively.

4.1. Definition of the Components' Implementations

At first, we describe behaviour of cores of primitive components, i.e. the components' implementations, which have to be defined by developer of the system (see Section 3.2.4). Description of behaviour of the Core component's implementation is:

$$Core_{impl} \stackrel{def}{=} (s_0, s_1, p_{save}, p_{load})(val)Core'_{impl}\lfloor \texttt{undef}, p_{save}, p_{load}\rfloor$$

$$\begin{aligned} Core'_{impl} \stackrel{\Delta}{=} {} & (val, p_{save}, p_{load})(p_{save}(val').Core'_{impl}\lfloor val', p_{save}, p_{load}\rfloor \\ & + p_{load}(ret).(\overline{ret}\langle val\rangle \mid Core'_{impl}\lfloor val, p_{save}, p_{load}\rfloor) \end{aligned}$$

where process $Core_{impl}$ can save a message received via name p_{save} and load the saved message and send it as a reply on a request received via name p_{load}.

Description of behaviour of the Session component's implementation is the following:

$$Session_{impl} \stackrel{def}{=} (s_0, s_1, r_{save}, r_{load}, p_{handle})Session'_{impl}\langle r_{save}, r_{load}, p_{handle}\rangle$$

$$\begin{aligned} Session'_{impl} \stackrel{def}{=} {} & (r_{save}, r_{load}, p_{handle})(save, load)(p_{handle}(ret) \\ & .\overline{ret}\langle save, load\rangle.Session'_{impl}\lfloor r_{save}, r_{load}, p_{handle}, save, load\rfloor) \end{aligned}$$

$$\begin{aligned} Session''_{impl} \stackrel{\Delta}{=} {} & (r_{save}, r_{load}, p_{handle}, save, load) \\ & (save(call).\overline{r_{save}}\langle call\rangle.Session'_{impl}\langle r_{save}, r_{load}, p_{handle}\rangle \\ & + load(call).\overline{r_{load}}\langle call\rangle.Session'_{impl}\langle r_{save}, r_{load}, p_{handle}\rangle) \end{aligned}$$

where process $Session_{impl}$ can receive via name p_{handle} an user's request, which is specified subsequently by inputs via names *save* or *load*, and pass it to process $Core_{impl}$ via names r_{save} or r_{load} (the required interfaces), respectively.

Finally, behaviour of the LOGIN component's implementation can be defined as follows:

$$\begin{aligned} Login_{impl} \triangleq\ & (s_0, s_1, p_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}) \\ & p_{init}(username, password, ret) \\ & .(Login_{verify}\langle username, password, ok, fail\rangle \\ & \mid Login'_{impl}\lfloor sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}, ret, ok, fail\rfloor \\ & \mid Login_{impl}\lfloor s_0, s_1, p_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}\rfloor) \end{aligned}$$

$$\begin{aligned} Login'_{impl} \triangleq\ & (sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}, ret, ok, fail)(new, d', t) \\ & (fail.\overline{ret}\langle \texttt{error}\rangle \ +\ ok.\overline{session_{clone}}\langle new\rangle.new(s'_0, s'_1, clone', r', p') \\ & .\overline{sys_{attach}}\langle s'_0, s'_1, d'\rangle.r'(r'^s_{save}, r'^s_{load}).p'(p'^g_{handle}) \\ & .\overline{core^g_{save}}\langle t\rangle.t(save).\overline{r'^s_{save}}\langle save\rangle.\overline{core^g_{load}}\langle t\rangle.t(load).\overline{r'^s_{load}}\langle load\rangle \\ & .p'^g_{handle}(handle).(\overline{s'_1} \mid \overline{ret}\langle handle\rangle)) \end{aligned}$$

where process $Login_{impl}$ can receive an user's initial request via name p_{init} as a triple of names (*username*, *password*, *ret*) and after successful verification of the user's name and password, the process returns a new session's handle via name *ret*. Name sys_{attach} provides an interface to attach new subcomponents into the system (see Section 3.2.2), name $session_{clone}$ is connected to a provided interface for cloning of SESSION component (see Section 3.2.3), and names $core^g_{save}$ or $core^g_{load}$ are connected to provided control interfaces for getting references to interfaces *save* or *load* of component CORE (see Section 3.2.1), respectively. The definition contains pseudo-application of process abstraction $Login_{verify}\langle username, password, ok, fail\rangle$, which represents description of behaviour of user's authentication process (e.g. $Login_{verify} \stackrel{def}{=} (\dots).\overline{ok}$ for authorising of all users).

4.2. Description of the Component Based System

Now, we can describe behaviour of individual components including their control parts, as well as behaviour and structure of a composite component, which represents the whole component based system. According to Section 3.2.4, behaviour of components CORE and SESSION can be described as follows:

$$\begin{aligned} Core \stackrel{def}{=}\ & (s_0, s_1, c, p^g_{save}, p^g_{load}).(p_{save}, p_{load}) \\ & (Ctrl_{Ifs}\langle p_{save}, p_{load}, p^g_{save}, p^g_{load}\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Core_{impl}\langle s_0, s_1, p_{save}, p_{load}\rangle) \end{aligned}$$

$$\begin{aligned} Session \stackrel{def}{=}\ & (s_0, s_1, c, r^s_{save}, r^s_{load}, p^g_{handle}).(r_{save}, r_{load}, p_{handle}) \\ & (Ctrl_{Ifs}\langle r_{save}, r_{load}, r^s_{save}, r^s_{load}, p_{handle}, p^g_{handle}\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Session_{impl}\langle s_0, s_1, r_{save}, r_{load}, p_{handle}\rangle) \end{aligned}$$

Behaviour of component LOGIN has to be described differently from the others, because it uses control interfaces sys_{attach}, $session_{clone}$, $core^g_{save}$, $core^g_{load}$, which can not be referenced (contrary to functional interfaces, see Section 2.2). This case can be compared with the description of *Update*

process in Section 3.2.3. The behaviour of component LOGIN can be described as follows:

$$\begin{aligned} Login \stackrel{def}{=}\ & (s_0, s_1, c, p^g_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}).(p_{init}) \\ & (Ctrl_{Ifs}\langle p_{init}, p^g_{init}\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\ & \mid Login_{impl}\lfloor s_0, s_1, p_{init}, sys_{attach}, session_{clone}, core^g_{save}, core^g_{load}\rfloor) \end{aligned}$$

Finally, behaviour and structure of a composite component, which represents the whole component based system, can be described as follows:

$$\begin{aligned} System \stackrel{def}{=}\ & (s_0, s_1, c, p^g_{init})(a, p_{init}, r'_{init}, r'^s_{init}) \\ & .(Ctrl_{Ifs}\langle p_{init}, p^g_{init}\rangle \mid Ctrl_{Ifs}\langle r'_{init}, r'^s_{init}\rangle \mid Ctrl_{EI}\langle p_{init}, r'_{init}\rangle \\ & \mid Ctrl_{clone}\lfloor c\rfloor \mid Ctrl_{SS}\langle s_0, s_1, a\rangle \mid System'\langle a, r'^s_{init}\rangle) \end{aligned}$$

$$\begin{aligned} System' \stackrel{def}{=}\ & (sys_{attach}, r^s_{init}) \\ & (p^g_{init}, core^g_{save}, core^g_{load}, sess^s_{save}, sess^s_{load}, sess^g_{handle}, login_{clone}, core_{clone}) \\ & (sess_{clone}, s_0^{login}, s_1^{login}, d^{login}, s_0^{core}, s_1^{core}, d^{core}, s_0^{sess}, s_1^{sess}, d^{sess}) \\ & (Login\langle s_0^{login}, s_1^{login}, login_{clone}, p^g_{init}, sys_{attach}, sess_{clone}, core^g_{save}, core^g_{load}\rangle \\ & \mid Core\langle s_0^{core}, s_1^{core}, core_{clone}, core^g_{save}, core^g_{load}\rangle \\ & \mid Session\langle s_0^{sess}, s_1^{sess}, sess_{clone}, sess^s_{save}, sess^s_{load}, sess^g_{handle}\rangle \\ & \mid \overline{sys_{attach}}\langle s_0^{login}, s_1^{login}, d^{login}\rangle \mid \overline{sys_{attach}}\langle s_0^{core}, s_1^{core}, d^{core}\rangle \\ & \mid \overline{sys_{attach}}\langle s_0^{sess}, s_1^{sess}, d^{sess}\rangle \mid \overline{p^g_{init}}\langle t\rangle.t(init).\overline{r^s_{init}}\langle init\rangle) \end{aligned}$$

5. Related Work

There have been proposed several component models [8]. In this section, we focus on two contemporary component models supporting some features of dynamic architectures and formal descriptions.

5.1. Fractal

The *component model Fractal* [3] 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 [2]. 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.

5.2. SOFA and SOFA 2.0

In the *component model SOFA* [12], 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) [9] for specification of components and *behaviour protocols* (BPs) for formal description of their behaviours. The BPs [21] 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* [5] 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.

6. Discussion and Future Work

The component model proposed in this paper is able to handle mobile architectures, unlike the SOFA that supports only a subset of dynamic architectures (implementing the update operation) or the Fractal/Fractive, which does not support components mobility. As is described in

Section 3.2, the π-calculus provides fitting formalism for description of software systems based upon the component model.

The proposed semantics of the component model permits to combine control interfaces and functional interfaces inside individual primitive components where the control actions can be invoked by the functional actions, i.e. by a system's business logic represented by business oriented services. This allows to build systems where functional (business) requirements imply changes of the systems' architectures. Regardless, in some cases, this feature can lead to *architectural erosion* and *architectural drift* [11], i.e. unpredictable evolution of the system's architecture. For that reason, the component model forbids dynamic changes of connections between control interfaces, which reduces architecture variability to patterns predetermined at a design-time. Formal description of the components integrating the control and functional actions can be compared with the transducer in the Fractal/Fractive approach (see Section 5.1).

The next feature of the component model is partially independence of a component's specification from its implementation (see the description of entities `CompAbstraction` and `CompImplementation` in Section 2.1). 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. (for comparison, see Section 5.1 and Section 5.2)

The attentive reader will have noticed that the process algebra π-calculus, as it is defined in Section 3.1 and applied to the formal description of behaviour of the component model's entities in Section 3.2, allows to describe only synchronous communication. Although, in most cases, we need to apply the component model to distributed software systems with asynchronous communication. This limitation is a consequence of the reduction relation's definition (see Definition 6 in Section 3.1). The problem can be solved by proposing of a "buffered" version of communication between interfaces (i.e. in process $Wire$ from Section 3.2.1) or, alternatively, by using of an asynchronous π-calculus [16].

The next important extension of the presented approach is application of typed π-calculus [10, 16], which allows to distinguish types of names. This feature is necessary to formally describe constraints of the type system of interfaces in behaviour of components. In the component model's metamodel, the type system is defined by instances of entity `TypOfInterface` and its descendants and related entities (see Section 2.1).

However, the above mentioned modifications are out of scope of this paper and a final version of the component model's formal description including the proposed extensions is part of current work. Further ongoing work is related to the realisation of a supporting environment, which allows integration of the component model into software development processes, including integration of verification tools and implementation support. The idea is to use results of the *ArchWare project* [1], especially for theorem-proving and model-checking[9]. We intend to use the *Eclipse Modeling Framework* (EMF) [4, 19] for modeling and code generation of tools based on the component model and the *Eclipse Graphical Modeling Framework* (GMF) [18] for developing graphical editors according to the rules described in the component model's metamodel (based on EMF).

7. Conclusion

In this paper, we have presented an approach, which contributes to specify component-based software systems with features of dynamic and mobile architectures. The proposed component model splits a software system into primitive and composite components according to decomposability of its parts, and the components' functional and control interfaces according to the types of required or provided services. The components can be described at different levels of abstraction, as their specifications and implementations.

[9] See the tools presented in documents D3.5b and D3.6c at [1].

Semantics of the component model's entities is formally described by means of the process algebra π-calculus (known as a calculus of mobile processes). Formal description of behaviour of a whole system can be derived from the visible behaviour of its primitive components and their compositions and communication, both defined at a design-time. The result is a π-calculus process, which describes the system's architecture, including its evolution and component mobility, and communication behaviour of the system. Thereafter, critical properties of the system can be verified by means of π-calculus model checker.

We are currently working on extending our approach to use asynchronous communication between components and a type system for their interfaces. Future work is related to integration of the component model into software development processes, including application of verification tools and implementation support. In the broader context, the research is a part of a project focused on formal specifications and prototyping of distributed information systems.

Acknowledgements This research has been supported by the Research Plan No. MSM 0021630528 "Security-Oriented Research in Information Technology".

References

[1] ArchWare project. http://www.arch-ware.org/, Nov. 2006.

[2] T. Barros. *Formal specification and verification of distributed component systems*. PhD thesis, Université de Nice – INRIA Sophia Antipolis, Nov. 2005.

[3] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium, Feb. 2004.

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, Aug. 2003.

[5] T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006*, Seattle, USA, 2006. IEEE Computer Society.

[6] J. Král and M. Žemlička. Autonomous components. In *SOFSEM 2000: Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*. Springer, 2000.

[7] J. Král and M. Žemlička. Software confederations and alliances. In *CAiSE Short Paper Proceedings*, volume 74 of *CEUR Workshop Proceedings*, pages 229–232. CEUR-WS.org, 2003.

[8] K.-K. Lau and Z. Wang. A survey of software component models (second edition). Pre-print CSPP-38, School of Computer Science, University of Manchester, Manchester, UK, May 2006.

[9] V. Mencl. Component definition language. Master's thesis, Charles University, Prague, 1998.

[10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Journal of Information and Computation*, 100:41–77, Sept. 1992.

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[12] 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.

[13] 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.

[14] M. Rychlý. Component model with support of mobile architectures. In *Information Systems and Formal Models*, pages 55–62. Faculty of Philosophy and Science in Opava, Silesian University in Opava, Apr. 2007.

[15] 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*. Faculty of Information Technology BUT, 2006.

[16] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, First paperback edition, Oct. 2003.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, second edition, Nov. 2002.

[18] The Eclipse Foundation. Eclipse Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/, Sept. 2007.

[19] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/, Sept. 2007.

[20] Unified Modeling Language, version 1.5. Document formal/03-03-01, Object Management Group, 2003.

[21] S. Višňovský. *Modeling software components using behavior protocols*. PhD thesis, Dept. of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002.