Behavioural Modeling of Services: from Service-Oriented Architecture to Component-Based System

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. Service-oriented architecture (SOA) is an architectural style for software systems' design, which merges well-established software engineering practices. There are several approaches to describe systems and services in SOA, the services' derivation, mutual cooperation to perform specific tasks, composition, etc. However, those approaches usually end up at the level of individual services and do not describe underlying systems of components, which form the services' implementation. This paper deals with formal description of behaviour of services in context of SOA and their decomposition into component systems with particular features such as dynamic reconfiguration and component mobility.

Key words: Service-oriented architecture, Component-based systems, Formal description, π-calculus

1 Introduction

Design of current information systems has to respect many functional and non-functional requirements, which have significant impact on software architectures. The requirements can include geographical and organisational limitations, support of distributed activities, integration of well-established (third party) software products, connection to a variable set of external systems, etc. Service-oriented architecture (SOA) seems to provide a solution [1]. The SOA is a widely used architectural style for design of distributed software systems, which merges well-established software engineering practices.

There are several approaches to describe information systems and services in SOA [2, 3]. Those approaches cover the whole development process from an analysis where individual services are derived from user requirements (usually represented by a system of business processes) to an implementation, which uses particular technologies implementing the services (e.g. Web Services). During this process, developers have to deal with description of a mutual cooperation of services to perform specific tasks, their composition, deployment, etc.

However, current approaches to SOA design usually end up at the level of individual services. They do not describe underlying systems of components,

which form design of individual services as component-based software systems with well-defined interfaces and behaviour.

This paper deals with formal description of services as component based systems (CBS) with particular features such as dynamic reconfiguration and component mobility in aspects of SOA. The *dynamic reconfiguration* represents creation, destruction and updating of components and their interconnections during the systems' run-time, while the *component mobility* allows creation of copies of components and changes of their context.

The remainder of this paper is organised as follows. In next parts of Section 1, we introduce SOA in Section 1.1 and CBS in Section 1.2 in more detail to outline their differences and briefly present a process algebra π-calculus in Section 1.3. In Section 2, we provide a formal description of services by means of the π-calculus in connection with a formal description of component-based systems as specified in following Section 3. In Section 4, we review main approaches that are relevant to our subject and discuss advantages and disadvantages of our approach compared with the reviewed approaches. To conclude, in Section 5, we summarise our approach, current results and outline the future work.

1.1 Service Oriented Architecture

Service-oriented architecture (SOA, [4]) is an architectural style for aligning business and IT architectures. It is a complex solution for analysis, design, maintaining and integration of enterprise applications that are based on services. *Services* are defined as autonomous platform-independent entities enabling access to their capabilities via their provided interfaces. They can communicate:

(a) **by passing data** between two services – in *service contracts*, services receiving the data are *requesters*, while services sending the data are *providers*,
(b) **by coordinating an activity** between two or more services – a multi-party collaboration between services that is usually known as *service choreography*.

There are also *service brokers* (*service registries*, e.g. UDDI registries) storing information about available service providers for potential service requesters.

The passing data between services can be implemented in different ways. We can distinguish the following styles of services implementation:

- **remote procedure calls (RPC)** where the emphasis is on services' interfaces with strictly defined properties determining their compatibility (e.g. SOAP, JSON-RPC and XML-RPC),
- **resource oriented services** where predefined interfaces are independent on actual type of transferred resources, objects represented by unique identifiers, so that each of them is interacted with in the same way (e.g. REST),
- **syndication-style publishing** where interfaces respect given standards for capturing all messages (e.g. Atom Publishing Protocol and RSS),
- **vendor-specific services** where generic RPC capabilities are difficult to use (e.g. Oracle Database SOAP).

A system that applies SOA can be described at three levels of abstraction:

Business processes describe the system as a hierarchically composed business process, where each decomposable process (at each level of the composition) represents sequence of steps in accordance with some business rules leading to a business aim[1].

Services implement business processes and their parts with well-defined interfaces and interoperability for the benefit of the business. *Business services* encapsulate distinct sets of business logic, *utility services* provide generic, non-application specific and reusable functionality, and *controller services* act as parent services to service composition members and ensure their assembly and coordination to the execution of the overall business task [4].

Components are implementation of services as CBSs with well-defined structure and description of its evolution for the benefit of the implementation.

1.2 Component Based Development and Systems

Component-based development (CBD, see [5]) is a software development methodology strongly oriented on composability and re-usability in software architecture. In the CBD, from a structural point of view, a *component-based system* (CBS) is composed of *components*, which are self contained entities accessible through well-defined *interfaces*.

The components can be primitive or composite. The *primitive components* are realised directly, beyond the scope of architecture description (they are "black-boxes"). The *composite components* are decomposable on systems of subcomponents at the lower level of architecture description (they are "grey-boxes"). This composition forms *a component hierarchy*.

A connection of compatible interfaces of cooperating components is realised via their *bindings* and actual organisation of interconnected components is called *configuration*. Syntax, semantics and composition of components can be defined by *component models* that are specific meta-models of software architectures supporting the CBD.

In our approach, we can distinguish the following types of components' interfaces according to functional aspects

functional interfaces for business-oriented services required or provided by a component with input and output parameters, respectively,

control interfaces for obtaining references to a component's provided functional interfaces or a component's fresh copy, binding a component's required functional interfaces to referenced provided functional interfaces of another component, or changing of behaviour and structure (e.g. adding of a new component as a composite component's subcomponent and its removing),

reference interface for passing of references to components or references to interfaces, which is required to support component mobility.

[1] business requirements are traditionally specified by a *business process model* (BPM)

	SOA	CBD/CBS
communication of entities	various forms of data passing (RPC, resources, etc.)	message passing via bindings of compatible interfaces
architecture of a system	service contracts on demand (via service brokers)	given by actual configuration, dynamic reconfiguration and mobility
composition of entities	business, utility and controller services	hierarchic composition (primitive and composite components)
compatibility of interfaces	by description of an interface and a type of communication	by behaviour of a component and specification of its interface
statefulness, statelessness	a service should not have externally visible state	a state can be given by and can affect behaviour/structure of a component

Table 1. The comparison of Service Oriented Architecture (SOA) and Component Based Development and Systems (CBD/CBS).

Unlike the services in SOA, components in CBD/CBS do not respect any business rules or aims, but only an initial configuration of a system, component hierarchy (encapsulation) and components' behaviour. Table 1 compares the features of SOA and CBD/CBS in aspects of communication of entities, description of their interconnection (a system's architecture), composition of entities, compatibility of their interfaces and visibility of their states (statefulness or statelessness of the entities). For detailed comparison, see [6].

1.3 Formal Basis

To describe services in SOA and CBS in formal way, we use the *process algebra* π*-calculus*, known also as a *calculus of mobile processes* [7], which is an extension of Robin Milner's *calculus of communicating systems* (CCS). The π-calculus allows modelling of systems with dynamic communication structures (i.e. mobile processes) by means of two concepts: processes and names. The *processes* are active communicating entities, primitive or expressed in π-calculus (denoted by uppercase letters in expressions), while the *names* are anything else, e.g. communication links (known as "ports"), variables, constants (data), etc. (denoted by lowercase letters in expressions). Processes use names (as communication links) to interact, and pass names (as variables, constants, and communication links) to another processes by mentioning them in interactions. Names received by a process can be used and mentioned by it in further interactions (as communication links). For the following description we suppose basic knowledge of the fundamentals of the π-calculus, a theory of mobile processes, according to [8].

- $\overline{x}\langle y\rangle.P$ is an *output prefix* that can send name y via name x (i.e. via the communication link x) and continue[2] 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 of name z in the process,

[2] it ensures process P can not proceed until a capability of the prefix has been exercised

- $P + P'$ 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,
- $\prod_{i=1}^{m} P_i = P_1 \mid P_2 \mid \ldots \mid P_m$ is a *multi-composition* of processes $P_1, \ldots, P_m$, for $m \geq 3$, which can proceed independently interacting via shared names,
- $(z)P$ is a *restriction* of the scope[3] of name z in process P,
- $(\tilde{x})P = (x_1, x_2, \ldots, x_n)P = (x_1)(x_2)\ldots(x_n)P$ is a *multi-restriction* of the scope of names $x_1, \ldots, x_n$ to process P, for $n \geq 2$,
- $!P$ 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 processes can be parametrised. A parametrised process, referred as an *abstraction*, is an expression of form $(x).P$.

When abstraction $(x).P$ is applied to argument y it yields process $P\{y/x\}$, i.e. process P with y substituted for every free occurrence of x. Application is the destructor of the abstraction. We can define two types of application: pseudo-application and constant application.

Pseudo-application $F\langle y \rangle$ of abstraction $F \stackrel{def}{=} (x).P$ is only an abbreviation of substitution $P\{y/x\}$. On the contrary, the *constant application* is a real syntactic construct. It allows to reduce a form of process $K\lfloor y \rfloor$, sometimes referred as an *instance* of process constant K, according to a *recursive definition* of process constant $K \stackrel{\Delta}{=} (x).P$. The result of the reduction yields process $P\{y/x\}$.

2 Formal Description of Services' Behaviour

In this section, we describe an approach to formal description of a service as a component-based system. In such description, the service has to be presented in two views (see the three levels of abstraction in Section 1.1):

1. *the service is a part of* SOA *architecture* and has description and relations to neighbouring services (requesters and providers) in a software system,
2. and *the service is a* CBS *system* with external and internal interfaces accessible by neighbouring systems and by internal components, respectively.

The first view is described in Section 2.1 and the second in Section 2.2.

2.1 Service as a Part of Service Oriented Architecture

Let's assume a service will be described as a part of SOA architecture of a software system (see Section 1.1). In Unified Modeling Language (UML), the service can be modeled by means of a stereotype "service" extending a class component in a class diagram [3]. The service interacts with its environment via

[3] the scope of a restriction may change as a result of interaction between processes

its *interfaces*, which are described as methods of the "service" class and by a sequence diagram (for an example, see Figure 1).

In the π-calculus, *a general service* `Service` with interfaces $i_1, \ldots, i_n$ can be described as a process abstraction

$$Service \stackrel{def}{=} (i_1, \ldots, i_n).(s_1, \ldots, s_m) \\ (Svc_{init}\langle i_1, \ldots, i_n, s_1, \ldots, s_m\rangle. \prod_{j=1}^{n} Svc_j \lfloor i_j, s_1, \ldots, s_m \rfloor)$$

where the pseudo-application of Svc_{init} initiates the service and the constant application of Svc_j, for each $j \in \{1, \ldots, n\}$, is responsible for specific processing of the service's interface i_j and can communicate with the others via shared names $s_1, \ldots, s_m$.

A service broker stores information about available service providers for potential service requesters, e.g. as references to the providers' interfaces. Its behaviour can be described as a π-calculus process abstraction $Broker$ as follows

$$\begin{aligned} Broker &\stackrel{def}{=} (a, g).(p)(Add\lfloor p, a\rfloor \mid Get\lfloor p, g, a\rfloor) \\ Add &\stackrel{\Delta}{=} (t, a).a(m, d).(t')(Add\lfloor t', a\rfloor \mid \overline{t}\langle t', m, d\rangle) \\ Get &\stackrel{\Delta}{=} (h, g, a).h(h', m, d).(\overline{g}\langle m\rangle.(Get\lfloor h', g, a\rfloor \mid \overline{a}\langle m, d\rangle) \ + \ d) \end{aligned}$$

where names representing the providers' interfaces (denoted by m internally) are stored via name a and retrieved back via name g, which are subsequently handled by constant applications of Add and Get, respectively. Special names (denoted by d internally) stored together with the names representing the providers' interfaces can be used later to remove these interfaces and do not provide them to potential service requesters anymore.

An Example: Figure 1 presents SOA services `ProcessPurchaseOrder`, `RequestGoods` and `RequestShipping` by means of an UML class diagram and an UML sequence diagram (for details, see [3]).

Each service is described by its interface and relations to neighbouring services. Service `ProcessPurchaseOrder` processes a purchase order. At the beginning, goods are requested in relevant amounts according to the purchase order via service `RequestGoods`. After that, service `RequestShipping` prepares a shipping request of resulting packages to a customer according to the purchase order. Finally, the shipping information are processed and a confirmation of the order is created and returned by service `ProcessPurchaseOrder` as a reply to the request.

Now, we can describe services `ProcessPurchaseOrder`, `RequestGoods` and `RequestShipping` as process abstractions PPO, RG and RS, respectively.

$$\begin{aligned} PPO &\stackrel{def}{=} (ppo, get_{RG}, get_{RS}).(get_{RG}(rg).get_{RS}(rs).PPO_{impl}\lfloor ppo, rg, rs\rfloor) \\ RG &\stackrel{def}{=} (rg, set_{RG}).(d)(\overline{set_{RG}}\langle rg, d\rangle.RG_{impl}\lfloor rg\rfloor) \\ RS &\stackrel{def}{=} (rs, set_{RS}).(d)(\overline{set_{RS}}\langle rs, d\rangle.RS_{impl}\lfloor rs\rfloor) \end{aligned}$$

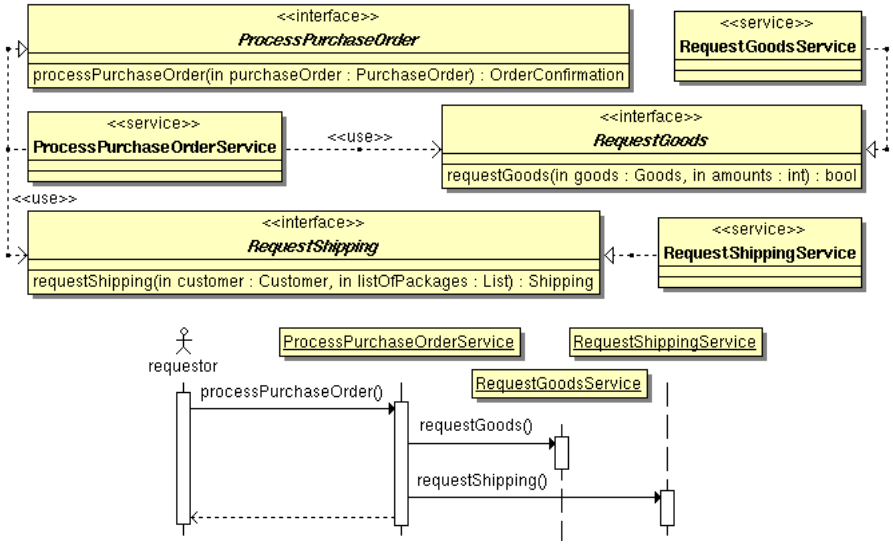

Fig. 1. A service's description in context of SOA (identified services, their interfaces and connections in a class diagram and their communication in a sequence diagram).

where ppo, rg and rs are names representing interfaces provided by the services, which are subsequently processed by applications of process constants PPO_{impl}, RG_{impl} and RS_{impl}, respectively. Process of initialisation of each individual service is described "in-line" before the application of a specific process constant and it uses names get_{RG}, get_{RS}, set_{RG} and set_{RS} as connections to service brokers' processes for storing and retrieving of `RequestGoods` and `RequestShipping` providers' interfaces, respectively.

Then, the whole system of the interconnected communicating services including their brokers can be described as follows

$$\begin{aligned} System \stackrel{def}{=} & (ppo).(get_{RG}, set_{RG}, get_{RS}, set_{RS}) \\ & (PPO\langle ppo, get_{RG}, get_{RS}\rangle \mid (rg)RG\langle rg, set_{RG}\rangle \mid (rs)RS\langle rs, set_{RS}\rangle \\ & \mid Broker\langle set_{RG}, get_{RG}\rangle \mid Broker\langle set_{RS}, get_{RS}\rangle) \end{aligned}$$

For testing purposes (e.g. to verify assembly relationships of the services), we may need to finish π-calculus description of the system and its services without knowledge of underlying implementation (as "a blackbox"). In such case, we can describe processing of the services' interfaces shortly as follows

$$\begin{aligned} PPO_{impl} &\stackrel{\Delta}{=} (ppo, rg, rs).ppo(po, r_{ppo}).\overline{rg}\langle g, a, r_{rg}\rangle.r_{rg}.\overline{rs}\langle c, lop, r_{rs}\rangle.r_{rs}.\overline{r_{ppo}} \\ RG_{impl} &\stackrel{\Delta}{=} (rg).rg(g, a, r_{rg}).\overline{r_{rg}} \\ RS_{impl} &\stackrel{\Delta}{=} (rs).rs(c, lop, r_{rs}).\overline{r_{rs}} \end{aligned}$$

2.2 Service as a Component Based System

In this section, we describe a service's underlying implementation as a CBS system (i.e. the service's behaviour and internal structure). The service can be represented by a system of components with external interfaces matching the services' provided interfaces. The following description is based on our previous research on distributed information systems as systems of asynchronous concurrent processes [9], features of mobile architectures in such systems [10, 11] and component models with support of mobile architectures and formal description.

Let's assume we have a general service `Service` with interfaces $i_1, \ldots, i_n$ described in the Section 2.1 as a process abstraction

$$Service \stackrel{def}{=} (i_1, \ldots, i_n).(s_1, \ldots, s_m) \\ (Svc_{init}\langle i_1, \ldots, i_n, s_1, \ldots, s_m\rangle. \prod_{j=1}^{n} Svc_j\lfloor i_j, s_1, \ldots, s_m\rfloor)$$

We focus on description of process constants $Svc_j(i_j,s_1,\ldots,s_m)$, for each $j \in \{1, \ldots, n\}$, which communicate via shared names $s_1, \ldots, s_m$ and are responsible for specific processing of the service's interfaces i_j. For the purpose of CBD, we can particularise the description as a process abstraction

$$Svc'_j \stackrel{def}{=} (i, s_{p_1}, \ldots, s_{p_k}, s_{r_1}, \ldots, s_{r_{(m-k)}}).Svc_j\lfloor i, s_1, \ldots, s_m\rfloor$$

where $p_1, \ldots, p_k, r_1, \ldots, r_{(m-k)} \in \{1, \ldots, m\}$ and $k \leq m$ and sets $\{s_{p_1}, \ldots, s_{p_k}\} \cap \{s_{r_1}, \ldots, s_{r_{(m-k)}}\} = \emptyset$ and $\{s_{p_1}, \ldots, s_{p_k}\} \cup \{s_{r_1}, \ldots, s_{r_{(m-k)}}\} = \{s_1, \ldots, s_m\}$. Names $s_{p_1}, \ldots, s_{p_{(m-k)}}$ and name i stand for "provided" interfaces as a selection of the service's provided shared names and its interface, respectively, while names $s_{r_1}, \ldots, s_{r_k}$ stand for "required" interfaces as the rest of required shared names.

Then, the service can be described as a CBS system (a component) with provided functional interfaces $i, s_{p_1}, \ldots, s_{p_{(m-k)}}$ and required functional interfaces $s_{r_1}, \ldots, s_{r_k}$ as it is shown is Section 3[4].

The Example: We describe an implementation of service `ProcessPurchase-Order`, which is the main service of the example mentioned in Section 2.1.

The service has been described as π-calculus process abstraction PPO where *ppo* represents an interface provided by the service and get_{RG} and get_{RS} represent connections to service brokers' processes for retrieving interfaces of `Request-Goods` and `RequestShipping`, respectively. The service's interface *ppo* is handled by process constant PPO_{impl} together with shared names rg and rs connected to processes representing actual providers of `RequestGoods` and `RequestShipping`.

Then, the service can be described as a CBS system (a component) with provided functional interface *ppo* and required functional interfaces rg and rs.

[4] The other two types of components' interfaces, i.e. *control interfaces* and *reference interfaces* (see Section 3), are not used in the description at the level of SOA.

3 Formal Description of a Component Based System

Now, we are ready to precisely describe a CBS system, which implements a service's behaviour and internal structure. The CBS system is defined by its initial configuration, component hierarchy and components' behaviour.

A *primitive component* is realised as "a black-box", which behaviour has to be formally defined by its developer as a π-calculus process where names represent the component's interfaces. The process 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 and structure 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 including their distribution to the component's subcomponents, etc.).

A whole CBS can be described as one component with provided and required interfaces, which represent the system's input and output actions, respectively.

Connections can interconnect only interfaces of the same types and dynamic creation of new connections and destruction of existing connection are permitted only for functional interfaces. Combining of actions of functional interfaces with actions of control interfaces is permitted only inside primitive components. This allows to build a system where functional (business) requirements imply changes of the system's architectures.

3.1 Notation of Names

Before we define and describe π-calculus processes implementing behaviour of a component and its 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: $s_0, s_1, r_1^s, \ldots, r_n^s, p_1^g, \ldots, p_m^g$ of any 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 r_i^s – a component accepts a request for binding given provided functional interface (included in the request as the interface's reference) to required functional interface r_i,
via p_j^g – a component accepts a request for referencing to 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.

3.2 Interface's References and Binding

At first, we define the auxiliary process $Wire$, 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).

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

Passing of messages from x to y by process $Wire$ is synchronous, which means that a message can not be received on x until a previously received message was successfully sent via y. To allow *asynchronous communication*, we need to use process $Wire_{async}$, which is a "buffered" version of process $Wire$.

$$\begin{aligned} Wire_{async} &\triangleq (x, y, d).(p)(Push\lfloor p, x, d\rfloor \mid Pop\lfloor p, y\rfloor) \\ Push &\triangleq (t, x, d).(x(m).(t')(Push\lfloor t', x, d\rfloor \mid \overline{t}\langle t', m\rangle) \ + \ d) \\ Pop &\triangleq (h, y).h(h', m).\overline{y}\langle m\rangle.Pop\lfloor h', y\rfloor \end{aligned}$$

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 funcional provided interfaces of another local components. Process $Ctrl_{Ifs}$ implementing the control interfaces can be defined as follows

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

where names $r_1, \ldots, r_n$, $r_1^s, \ldots, r_n^s$, $p_1, \ldots, p_m$, $p_1^g, \ldots, p_m^g$ have been defined at the beginning of Section 3.1, processes $SetIf$ and $GetIf$ allow to bind required interfaces and to get references to provided interfaces, respectively, and process $Plug$ is an auxiliary process.

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 $Ctrl_{EI}$.

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

3.3 Control of a Component's Life-cycle

Control of a composite component's life-cycle[5] can be described as process $Ctrl_{SS}$.

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

where names s_0 and s_1 represent the component's interfaces that accept stop and start requests, respectively. The requests for stopping and starting the component are distributed to its subcomponents via names p_0 and p_1, respectively, as it is described in processes $Life$ and $Dist$[6]. Name a of process $Ctrl_{SS}$ can be used to attach a new subcomponent's stop and start interfaces (at one step), i.e. to connect them to the relevant composite component's stop and start interfaces via names p_0 and p_1 and via processes $Wire$, as it is described in process $Attach$. Third name, which is received via name a, can be used later to detach the subcomponent's previously attached stop and start interfaces.

3.4 Component Behaviour of 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

[5] a primitive component handles stop and start interfaces directly

[6] in the initial state, the component and its subcomponents are stopped

$s_0, s_1, r_1, \ldots, r_n, p_1, \ldots, p_m$ 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). 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 3.1.

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

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

where processes $Ctrl_{Ifs}$ represent behaviour of control parts of components related to their interfaces (see Section 3.2), process $Ctrl_{SS}$ represents behaviour of a component's control part handling its stop and start requests (see Section 3.3), and process $Ctrl_{EI}$ describes behaviour of communication between internal and external functional interfaces of a component (see Section 3.2).

4 Related Work and Discussion

Approaches to modeling of services can be broadly divided into two groups. In the first group, there are *approaches to describe information systems and services in SOA* [2, 3], which cover the whole development process, but usually end up at the level of individual services and do not describe their underlying implementation as CBS systems. The second group contains *several component models*[7] [12], which can bring features of SOA into CBD so that SOA becomes a specific case of a component model with dynamic reconfiguration [13]. In SOA, developers have to deal with description of a mutual cooperation of services to perform specific tasks, their hierarchical composition, deployment, etc., as well as with description of the underlying implementation.

In addition to those approaches, we should mention of *Service Component Architecture* (SCA, [14]), which aims at providing of a component model for

[7] i.e. meta-models specifying systems of architectural entities, their properties, styles of their interconnections, and rules of evolution of the architecture

SOA. SCA defines a model for assembling of service components and a model for creation of component-based services with reference implementation in Java, C++, BPEL, PHP, JavaScript, XQuery and SQL. However, SCA does not provide formal description of services' and components' behaviour and structure.

In this section, we focus mainly on two contemporary component models supporting features of SOA and formal description, SOFA 2.0 and Fractal.

Component model SOFA *2.0* [15] is aimed at removing several limitations of component model SOFA [16] – mainly the lack of support of dynamic reconfiguration, well-structured and extensible control parts of components, and multiple styles of communication between components. The original version of SOFA uses a *component definition language* (CDL) for architecture description (specification) of components and *behaviour protocols* (BPs) for formal description of their behaviour. It allows only a dynamic update of components during a system's runtime with compatibility check. The SOFA 2.0 introduces extended dynamic reconfiguration, which is predefined at design time by *reconfiguration patterns* [13]: nested factory (adds a new component or a new connection), component removal and utility interface patterns. Utility interfaces can be referred, references freely passed among components, which can establish connections using these references, independently of their level in architecture hierarchy. It brings into CBD features of SOA, which becomes a specific case of a component model where all components (services) are interconnected solely via their utility interfaces. To control dynamic reconfiguration, SOFA 2.0 introduces microcomponents and multiple communication styles. The *microcomponents* [17] are minimal primitive components designed to capture an architecture of a general component's controller parts, to express that the component's controller requires a certain control (micro)component and to specify interconnections of control and functional parts of the component. The *multiple communication styles* [15] define functionality of connectors (remote method invocation, message passing, streaming, and distributed shared memory), which can restrict binding of its interfaces and affect its runtime optimisation.

Component model Fractal [18] is a general component composition framework supporting dynamic architectures with components formed out of two parts: a controller ("a membrane" enclosing a component) and a content (primitive or composed of a finite number of nested components controlled by the controller). Behaviour of Fractal components can be formally described by means of *parametrised networks of communicating automata* language [19]. A primitive component is modelled as and formally described by means of a finite state *parametrised labelled transition system* (pLTS), while a composed component is defined using a *parametrised synchronisation network* (pNet, which is a pLTS computed as a product of subcomponents' pLTSs and a transducer synchronising actions of the corresponding pLTSs of the subcomponents). *Fractal WS* [6] provides connection between Fractal and Web Service technology (WS), which can be used to implement SOA. Any interface provided by a Fractal component can be transformed into a WS and any (external) WS can be accessed inside an assembly of Fractal components at any level of hierarchy using a dedicated

proxy component. Moreover, there exists *Fractal SCA* experiment [6] bridging component-based applications written in the Fractal and SCA technologies.

In comparison of our approach with the reviewed approaches, we can find many similar features, which are typical for description of component based systems' behaviour and structure. Contrary to the Fractal or SOFA 2.0, our approach describe services and components in the same way, but with respect to their differences (i.e. services are not components and vice versa, see Section 1.2). Moreover, we do not introduce a new formalism but use the well-established π-calculus to describe services in SOA architecture, as well as to define individual primitive components' behaviour and their composition into hierarchically structured CBS. This approach allows to utilise existing tool for model-checking of π-calculus processes and formal verification of their properties (e.g. Advanced/Another Bisimulation Checker [20] or tools from ArchWare Project [21]). However, our approach can have also drawbacks, e.g. complex description of primitive components' behaviour combining actions of functional interfaces with actions of control interfaces or insufficient visibility of a component based system's structure during its evolution. The evolution of the system means evolution (reduction) of π-calculus process representing too much tightly bound behaviour and structure.

5 Conclusion and Future Work

In this paper, we have presented an approach to formal description of behaviour and structure of services in SOA architecture as a CBS systems with features of dynamic and mobile architectures. We use calculus of mobile processes, the π-calculus, for specification of the services, definition of individual primitive components' behaviour and their composition into a hierarchically structured CBS system implementing the services' behaviour.

Future work is related to integration of the approach into visual UML modeling and CASE tools and automatic "top-down" and "bottom-up" generation of a formal description of SOA architectures and underlying CBS systems.

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

References

1. Constantinides, C., Roussos, G.: Service Patterns for Enterprise Information Systems. In: Service-Oriented Software System Engineering: Challenges and Practices. IGI Global, Hershey, PA, USA (April 2005) 201–225
2. Amsden, J.: Modeling SOA, parts I–V. IBM developerWorks (October 2007)
3. Rychlý, M., Weiss, P.: Modeling of service oriented architecture: From business process to service realisation. In: ENASE 2008 Third International Conference on Evaluation of Novel Approaches to Software Engineering Proceedings, Institute for Systems and Technologies of Information, Control and Communication (May 2008) 140–146

4. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (August 2005)
5. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. second edn. Addison Wesley Professional (November 2002)
6. Collet, P., Coupaye, T., Chang, H., Seinturier, L., Dufrêne, G.: Components and services: A marriage of reason. Technical Report ISRN I3S/RR-2007-17-FR, Project RAINBOW, CNRS (May 2007)
7. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Journal of Information and Computation **100** (September 1992) 41–77
8. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. New Ed edn. Cambridge University Press (October 2003)
9. Rychlý, M.: Towards verification of systems of asynchronous concurrent processes. In: Proceedings of 9th International Conference Information Systems Implementation and Modelling (ISIM'06), MARQ (April 2006) 123–130
10. Rychlý, M., Zendulka, J.: 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 (October 2006) 206–213
11. Rychlý, M.: Component model with support of mobile architectures. In: Information Systems and Formal Models, Faculty of Philosophy and Science in Opava, Silesian university in Opava (April 2007) 55–62
12. Lau, K.K., Wang, Z.: 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)
13. Hnětynka, P., Plášil, F.: Dynamic reconfiguration and access to services in hierarchical component models. In: Proceedings of CBSE 2006. Volume 4063 of Lecture Notes in Computer Science., Springer (2006) 352–359
14. OSOA: SCA service component architecture: Assembly model specification. Technical Report SCA version 1.00, The Open SOA Collaboration (March 2007)
15. Bureš, T., Hnětynka, P., Plášil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: Proceedings of SERA 2006, Seattle, USA, IEEE Computer Society (August 2006) 40–48
16. Plášil, F., Bílek, D., Janeček, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: 4th International Conference on Configurable Distributed Systems, Los Alamitos, CA, USA, IEEE Computer Society (May 1998) 43–51
17. Mencl, V., Bureš, T.: Microcomponent-based component controllers: A foundation for component aspects. In: Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005), Taipei, Taiwan, IEEE Computer Society Press (December 2005) 729–737
18. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium (February 2004)
19. Barros, T.: Formal specification and verification of distributed component systems. PhD thesis, Université de Nice – INRIA Sophia Antipolis (November 2005)
20. Briais, S.: Abc – the Advanced Bisimulation Checker. http://lamp.epfl.ch/ sbriais/abc/ (March 2008)
21. ArchWare Consortium: ArchWare project. http://www-systems.dcs.st-and.ac.uk/wiki/ArchWare (March 2008)