A Case Study on Behavioural Modelling of Service-Oriented Architectures

Marek Rychlý

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

Abstract. 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. In this paper, we introduce a new approach to describe behaviour of services in SOA, including behaviour of underlying systems of components, which form the services' implementation. The behavioural description uses the process algebra π-calculus and it is demonstrated on a case study of a service-oriented architecture for functional testing of complex safety-critical systems.

Key words: Service-oriented architecture, Behavioural modelling, Formal description, π-calculus

1 Introduction

Service-oriented architecture (SOA) is a well-established 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. It represents a model in which functionality is decomposed into small, distinct units, "services", which can be distributed over a network and can be combined together and reused to create business applications [1]. A system that applies SOA can be described at three levels of abstraction: as a system of business processes, services, and components.

At the first level, the system is described as a hierarchically composed business process, where each decomposable process (at each level of the composition) represents a sequence of steps in accordance with some business rules leading to a business aim.

The business processes or their parts are implemented by *services*, which are defined as autonomous platform-independent entities enabling access to their capabilities via their interfaces. *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 [1].

Every service can be implemented as a *component-based system* (CBS) with well-defined structure and description of its evolution for the benefit of the implementation. Then, *components* are self contained entities, parts of component-based systems accessible through well-defined *interfaces* and interconnected and communicating via *bindings* of these interfaces. *Primitive components* are realised directly, beyond the scope of architecture description (they are "black-boxes"), while *composite components* are decomposable on systems of subcomponents at the lower level of architecture description (they are "grey-boxes").

This paper deals with formal description of behaviour of services and underlaying component-based systems, by means of process algebra π-calculus and with focus on dynamic aspects of SOA. The proposed approach is demonstrated on a case study of a service-oriented architecture for functional testing of complex safety-critical systems.

The remainder of this paper is organised as follows. In Section 1.1, we briefly describe the π-calculus to provide formal basis for our approach. The case study is introduced in Section 2 and described in more detail in Section 3 as a service-oriented architecture and in Section 4 as an underlying component-based system. Finally, Section 5 and Section 6 deal with formal description. In Section 7, the proposed approach is discussed and compared with current approaches relevant to our subject. To conclude, in Section 8, we summarise the contribution of this paper and outline the future work.

1.1 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* [2]. It 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, while the *names* are anything else, e.g. communication links (known as "ports"), variables, constants (data), etc. Processes use names (as communication links) to interact, and they pass names (as variables, constants, and communication links) to another processes by mentioning them in the interactions. Names received by a process can be used and mentioned by it in further interactions (as communication links). For description of our approach in this paper, we suppose basic knowledge of the fundamentals of the π-calculus, a theory of mobile processes, according to [3]:

- $\overline{x}\langle y\rangle.P$ is an *output prefix* that can send name y via name x (i.e. via the communication link x) and continue 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;
- $P + P'$ 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[1] 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 a 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 an abbreviation of substitution $P\{y/x\}$. On the contrary, the *constant application* is a real syntactic construct, which 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 Case Study Specification

As a case study, we adopt specification of a SOA for functional testing of complex safety-critical systems, more specifically *a testing environment of a railway interlocking control system*, which has been described in [4]. The environment allows to distribute and run specific tests over a wide range of different testing environments, varying in their logical position in the system's architecture.

The testing environment is described as a composition of a tester and a set of external system simulators. The *external system simulators* totally or partially represent and simulate a tested environment interacting with *system under testing* (SUT), e.g. a behaviour of field objects (points, track circuits, coloured signals, etc.). The *tester* automatically executes specific tests that are coded in *test scripts* and coordinates the SUT via a *man machine interface* (MMI) and the external system simulators. The SUT is represented by the *computer based control system* (CBCS), running the *control software*, interacting with operators by means of the MMI and monitoring or controlling *external systems* of rail yards by means of sensors or actuators, which are accessible via *external systems interface*. Each *rail yard* has its own instance of the testing environment with specific sensors and actuators where assigned tests are automatically executed. For detailed description, see [4].

To implement a system for distribution and execution of the tests over various instances of the testing environments, [4] proposes to use SOA. The system consists of a test manager, which is able to receive a test script and execute it

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

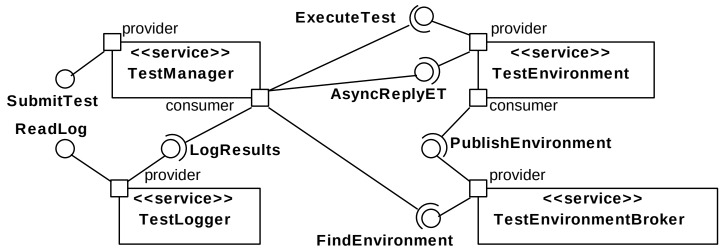

Fig. 1. Services of the testing environment and their interfaces (for notation, see [5]).

in an instance of the testing environment. Available testing environments are registered by a broker and provided to the test manager at its request.

3 Service Identification

From the description of the testing environment and the system's architecture, the following tasks can be identified as invocations of services: "Submit Test", "Execute Test", "Log Results", "Read Log", "Publish Environment", and "Find Environment". The tasks can be implemented by the following business (entity) services, as it is described in Figure 1: `TestManager`, `TestEnvironment`, `TestEnvironmentBroker`, and `TestLogger`.

At first, service `TestManager` receives a test script from a tester via its interface `SubmitTest`. Then, it calls `FindEnvironment` of service `TestEnvironmentBroker` to search for a testing environment that would be suitable for the test script. The broker, which has previously accepted a registration request from a specific service `TestEnvironment` via its interface `PublishEnvironment`, provides `TestManager` with a reference to the registered service as a return value of the call of `FindEnvironment`.

After that, service `TestManager` passes the test script to the referred service `TestEnvironment` via its interface `ExecuteTest`. When the test script is finished, service `TestEnvironment` forwards its results back to service `TestManager`, which logs the results via `LogResults` of service `TestLogger`. Those results can be viewed later via `ReadLog`, which is provided by service `TestLogger` to the tester.

Figure 2 shows a choreography of the services as an UML sequence diagram. Detailed description of the services as classes and their interfaces with relevant stereotypes is described in the UML class diagram in Figure 3. Service `TestEnvironment` is invoked asynchronously via `ExecuteTest`, i.e. a reply corresponding to the request will be returned later via the service's interface `AsyncReplyET`.

4 Component-Based System

Railway interlocking control systems are safety-critical systems and can be described as component-based systems [6]. A testing environment of such systems

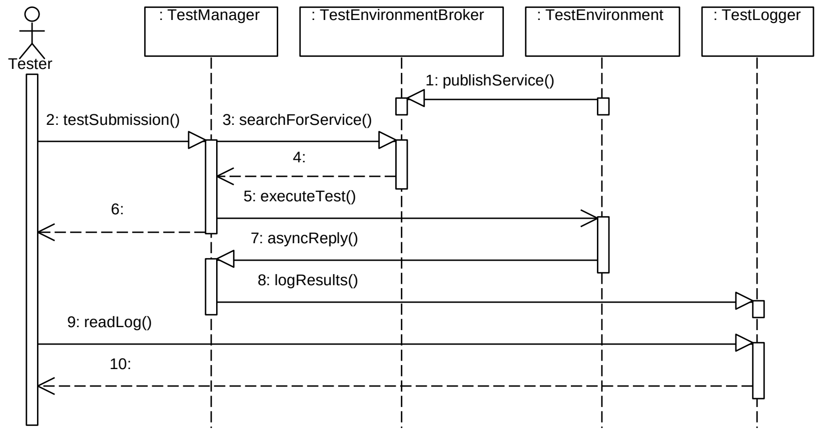

Fig. 2. The choreography of services in the testing environment.

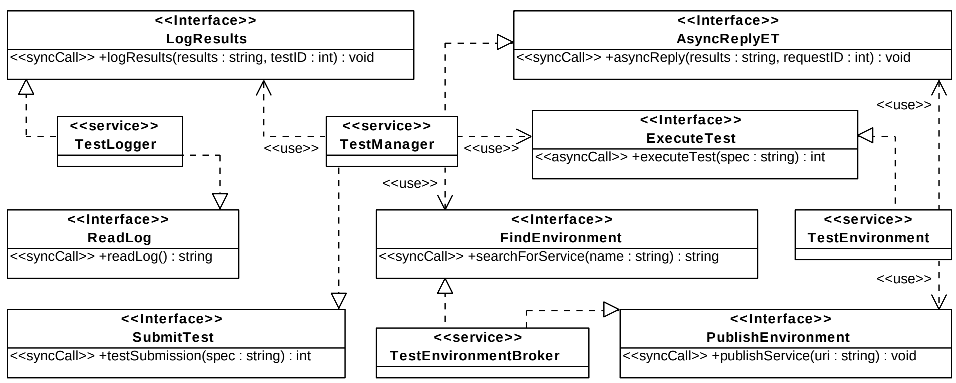

Fig. 3. Services of the testing environment as UML classes.

has to interact with the systems' components, as it is described in Section 2. For that reason, a part of the testing environment, which is directly connected to a system under testing (via the external systems simulators), has character of a component neighbouring to the system and can be described as CBS.

Figure 4 describes composite component testEnvironment, which represents service `TestEnvironment` from Section 3. The used notation is based on our component model [7] (it is not standard UML), whose detailed description is out of the scope of this paper. However, in this section, we try to outline the main ideas and informally describe structure of the composite component and behaviour of its subcomponents controller, environment, test and output.

Component testEnvironment receives a test script via provided interface `executeTest`, which is internally processed by component controller. The script is represented by a fresh component, which does required testing after binding of its interfaces to component environment.

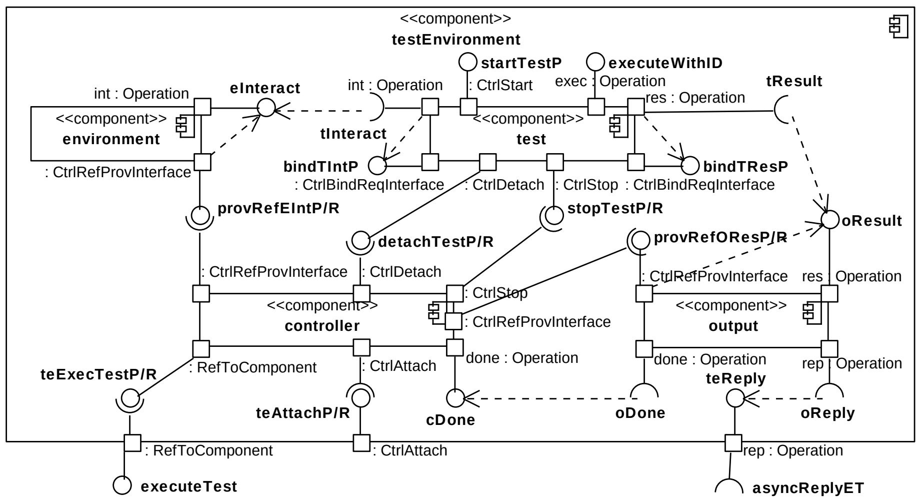

Fig. 4. Composite component TestEnvironment (a specific UML-like notation).

At first, component controller attaches the new component as a subcomponent test of component testEnvironment via its control interface `teAttachP`. Then, it binds interfaces `tInteract` and `tResult` of the new component to interface `eInteract` of component environment and interface `oResult` of component output, respectively. Finally, component test is activated via interface `startTestP` and executed with a new identifier via interface `executeWithID`. The identifier is also returned by component testEnvironment as a reply of the test script's submission.

Component test performs the test script by interacting with component environment via its interface `eInteract`. When the test script is finished, component test sends the test's results and its identifier to component output via its interface `oResult`. Then, component output notifies component controller via its interface `cDone` and forwards the results and the identifier out of the component testEnvironment via its external interface `asyncReplyET`.

After component controller is notified about the finished test script, it is able to receive and execute another test script, i.e. to attach a new component in the place of component test. Before that, component test with the old script is stopped via interface `stopTestP` and detached via control interface `detachTestP`[2].

5 Formal Description of the Services

In this section, we describe behaviour of the services in the testing environment. Behaviour of services `TestManager`, `TestEnvironmentBroker`, `TestEnvironment`, and

[2] In the diagram in Figure 4, only these two interfaces of test are connected with controller, because the rest of the test's interfaces are used only during its nesting and their connections do not exist outside of controller component.

`TestLogger` can be described by means of π-calculus process abstractions TM, TEB, TE, and TL, respectively. These process abstractions use names st, pe, fe, et, ar, lr, and rl as representations of the services' interfaces `SubmitTest`, `PublishEnvironment`, `FindEnvironment`, `ExecuteTest`, `AsyncReplyET`, `LogResults`, and `ReadLog`, respectively.

According to the description of `TestEnvironment` in Section 3, process abstraction TM describing behaviour of service `TestManager` is defined as follows:

$$\begin{aligned}
TM &\stackrel{def}{=} (st, fe, lr).(s)(TM_{st}\lfloor st, fe, s\rfloor \mid TM_{ar}\lfloor lr, s\rfloor) \\
TM_{st} &\stackrel{\Delta}{=} (st, fe, s).st(test, ret).(r, r') \\
&\quad (\overline{fe}\langle r\rangle.r(et', ar').\overline{et'}\langle test, r'\rangle.(r'(id).\overline{ret}\langle id\rangle \mid \overline{s}\langle ar'\rangle \mid TM_{st}\lfloor st, fe, s\rfloor)) \\
TM_{ar} &\stackrel{\Delta}{=} (lr, s).s(ar')ar'(res, id).\overline{lr}\langle res, id\rangle \mid TM_{ar}\lfloor lr, s\rfloor
\end{aligned}$$

where st, fe, and lr are names representing the service's interfaces and subsequently processed by constant applications of TM_{st} and TM_{ar}.

Constant application $TM_{st}\lfloor st, fe, s\rfloor$ receives a pair of names $(test, ret)$ from a client via name st. In the pair, name $test$ represents a submitted test script and name ret will be used later to send a return value to the client. Then, a request for a testing environment is sent via name fe and the environment as a reply is received via name r. Name et', which represents an interface `ExecuteTest` of the environment, is used to send $test$. Name id, which is received as a return value, is forwarded to the client, while name ar' is sent via shared name s into process constant TM_{ar}. Constant application $TM_{ar}\lfloor lr, s\rfloor$ receives name ar' via shared name s. After the test script is finished, name ar' is used to receive the test's result res and its id. These names, as a pair (res, id), are immediately sent via name lr.

Process abstraction TEB, which describes behaviour of service `TestEnvironmentBroker`, is defined as follows:

$$\begin{aligned}
TEB &\stackrel{def}{=} (pe, fe).(q)(TEB_{pub}\lfloor q, pe\rfloor \mid TEB_{find}\lfloor q, fe, pe\rfloor) \\
TEB_{pub} &\stackrel{\Delta}{=} (t, pe).pe(i, d).(t')(\overline{t}\langle t', i, d\rangle \mid TEB_{pub}\lfloor t', pe\rfloor) \\
TEB_{find} &\stackrel{\Delta}{=} (h, fe, pe).h(h', i, d).(TEB_{find}\lfloor h', fe, pe\rfloor \mid (\overline{fe}\langle i\rangle.\overline{pe}\langle i, d\rangle \;+\; d))
\end{aligned}$$

where pe and fe are names representing the service's interfaces `PublishEnvironment` and `FindEnvironment`, respectively, and subsequently processed by the constant applications of TEB_{pub} and TEB_{find}. By the composition of their constant applications with shared name q, process abstraction TEB implements basic operations on a simple queue (i.e. a First-In-First-Out (FIFO) data structure).

The application of process constant TEB_{pub} receives a pair of names (i, d) via name pe and creates a new name t'. Then, it proceeds as a composition of a constant application of $TEB_{pub}\lfloor t', pe\rfloor$, which handles future requests, and process $\overline{t}\langle t', i, d\rangle$, which enqueues the received pair (i, d) by sending them via name t, which is *the current tail of the queue*, together with name t', a new tail of the queue used in the future requests.

The application of process constant TEB_{find} dequeues a front item of the queue as a triple of names (h', i, d) via name h, which is *the current head of the queue*. Then, it proceeds as a composition of a constant application of $TEB_{find}\lfloor h', fe, pe \rfloor$, which handles future requests, and a sum of capabilities of process $\overline{fe}\langle i \rangle.\overline{pe}\langle i, d \rangle$, which provides name i as an interface for potential service requesters and enqueues it back to the queue via name pe, and process d, which, after receiving a name via name d, allows to remove the interface and does not provide it to potential service requesters anymore.

Behaviour of service `TestEnvironment` is described as process abstraction TE and defined as follows:

$$
\begin{aligned}
TE &\stackrel{def}{=} (et, ar, pe).TE_{init}\langle et, ar, pe \rangle.TE_{impl}\langle et, ar \rangle \\
TE_{init} &\stackrel{def}{=} (et, ar, pe).\overline{pe}\langle et, ar \rangle \\
TE_{impl} &\stackrel{def}{=} (et, ar).(s_0, s_1, ar^s, et^g) \\
&\quad (\overline{ar^s}\langle ar \rangle \mid (d, t)(\overline{et^g}\langle t \rangle.t(p).Wire\lfloor et, p, d \rfloor) \mid TE_{comp}\langle s_0, s_1, et^g, ar^s \rangle)
\end{aligned}
$$

where et, ar, and pe are names representing the service's interfaces `ExecuteTest`, `AsyncReplyET`, and `PublishEnvironment`, respectively. Initialisation of the service is described as process abstraction TE_{init}, which sends the service's interfaces represented by names et and ar via name pe (i.e. publishes the corresponding interfaces via interface `PublishEnvironment`). After the initialisation, names et and ar are processed by pseudo-application $TE_{impl}\langle et, ar \rangle$, which describes behaviour of a component-based system implementing the service (service `TestEnvironment` is implemented as the component-based system, see Section 4). Process abstraction TE_{comp} will be described later, in Section 6.

Finally, process abstraction TL, which describes behaviour of service `TestLogger`, is defined as follows:

$$
\begin{aligned}
TL &\stackrel{def}{=} (lr, rl).(s)(TL_{lr}\lfloor lr, s \rfloor \mid TL_{rl}\lfloor rl, s \rfloor) \\
TL_{lr} &\triangleq (lr, t).lr(res, id).(t')(\bar{t}\langle t', res, id \rangle \mid TL_{lr}\lfloor lr, t' \rfloor) \\
TL_{rl} &\triangleq (rl, h).h(h', res, id).rl(ret).\overline{ret}\langle res, id \rangle.TL_{rl}\lfloor rl, h' \rfloor
\end{aligned}
$$

where lr and rl are names representing the service's interfaces `LogResults` and `ReadLog`, respectively, and subsequently processed by the applications of process constants TL_{lr} and TL_{rl}. The process abstraction TL uses an internal queue to store log results. The queue is accessed in process constants TL_{lr} and TL_{rl} via name h for a head of the queue and name t for a tail of the queue, respectively. At the beginning, h and t are identical to name s in process abstraction TL.

Constant application $TL_{lr}\lfloor lr, t \rfloor$ receives a pair of names (res, id) via name lr, which will be added into the internal queue. It creates name t' (as a new tail of the queue) and sends via t' the pair of names (res, id) and name t (an original tail of the queue). Concurrently, the process proceeds as the application of process constant TL_{lr} with name t' (the new tail of the queue).

Constant application $TL_{rl}\lfloor rl, h \rfloor$ receives a first queued item via name h (from a head of the queue). This item contains a pair of names (res, id) and name h' (a new head of the queue). After the pair of names (res, id) is requested via name rl, it is sent via name ret as a reply and the process proceeds as the application of process constant TL_{rl} with name h' (the new head of the queue).

Behaviour of the whole system of the interconnected services can be described as process abstraction $System$, which provides names st and rl representing interfaces `SubmitTest` and `ReadLog`, respectively, and which is defined as follows:

$$\begin{aligned}System \stackrel{def}{=} \ &(st, rl).(et, ar, lr, pe, fe)\\ &(TM\langle st, fe, lr\rangle \mid TE\langle et, ar, pe\rangle \mid TL\langle lr, rl\rangle \mid TEB\langle pe, fe\rangle)\end{aligned}$$

6 Formal Description of the Component-Based System

All processes, which represent behavioural descriptions of individual services, have been described completely, except for process abstraction TE of service `TestEnvironment` implemented as a component-based system with behaviour described by pseudo-application $TE_{comp}\langle s_0, s_1, ar^s, et^g\rangle$. In this section, we describe behaviour of primitive components controller, environment, test, and output, as process abstractions Ctr, Env, $Test$, and Out, respectively, and their parent composite component testEnvironment, as process abstraction TE_{comp}.

6.1 Core Behaviour of Primitive Components

Core behaviour of primitive components output and controller can be defined as process abstractions Out_{core} and Ctr_{core}, respectively, as follows:

$$\begin{aligned}
Out_{core} &\stackrel{def}{=} (p_{oResult}, r_{oDone}, r_{oReply}).Out'_{core}\lfloor p_{oResult}, r_{oDone}, r_{oReply}\rfloor\\
Out'_{core} &\stackrel{\Delta}{=} (p_{oResult}, r_{oDone}, r_{oReply}).p_{oResult}(res, id).\overline{r_{oDone}}\langle id\rangle.\\
&\quad (\overline{r_{oReply}}\langle res, id\rangle \mid Out'_{core}\lfloor p_{oResult}, r_{oDone}, r_{oReply}\rfloor)\\
Ctr_{core} &\stackrel{def}{=} (p_{cDone}, p_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt},\\
&\qquad r_{provRefORes}).Ctr'_{core}\lfloor p_{cDone}, p_{teExecTest},\\
&\qquad r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}\rfloor\\
Ctr'_{core} &\stackrel{\Delta}{=} (p_{cDone}, p_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt},\\
&\qquad r_{provRefORes}).p_{teExecTest}(ts, ret).ts(r'_{stopTest}, r'_{startTest}, r', p').\\
&\quad \overline{r_{stopTest}}.\overline{r_{detachTest}}.\overline{r_{teAttach}}\langle r'_{stopTest}, r'_{startTest}, r_{detachTest}\rangle.\\
&\quad r'(p'_{bindTInt}, p'_{bindTRes}).p'(p'_{provRefExecuteWithID}).(ret')(\\
&\qquad \overline{r_{provRefEInt}}\langle ret'\rangle.ret'(eInteract).\overline{p'_{bindTInt}}\langle eInteract\rangle.\\
&\qquad \overline{r_{provRefORes}}\langle ret'\rangle.ret'(oResult).\overline{p'_{bindTRes}}\langle oResult\rangle.\\
&\qquad \overline{p'_{provRefExecuteWithID}}\langle ret'\rangle.ret'(p'_{executeWithID}).\overline{r'_{startTest}}.\\
&\qquad ((id)\overline{ret}\langle id\rangle.\overline{p'_{executeWithID}}\langle id\rangle.\overline{id} \mid p_{cDone}(id').id'.\\
&\qquad\quad Ctr'_{core}\lfloor p_{cDone}, p_{teExecTest}, r_{teAttach}, r_{detachTest},\\
&\qquad\qquad r'_{stopTest}, r_{provRefEInt}, r_{provRefORes}\rfloor)\ )
\end{aligned}$$

where the components' provided or required interfaces are represented by names $p_{\dots}$ or $r_{\dots}$, respectively, without the last character ($\dots P/R$, see Figure 4).

Process abstraction Out_{core} is defined as the constant application of Out'_{core}. It receives a pair of names (res, id) via name $p_{oResult}$ representing interface `oResultP`. Then, id is sent via name r_{oDone} (interface `oDoneR`) and (res, id) is forwarded via name r_{oReply} (interface `oReplyR`) out of the composite component.

Process constant Ctr'_{core}, which is applied by process abstraction Ctr_{core}, receives a pair of names (ts, ret) via name $p_{teExecTest}$. Moreover, via name ts, the constant receives also names $r'_{stopTest}$, $r'_{startTest}$, c, and indirectly also names $p'_{bindTInt}$, $p'_{bindTRes}$, and $p'_{provRefExecuteWithID}$, which represent interfaces of a new component compatible with component test and implementing a test script. Name ret will be used later to send an identifier of the test's results as a return value. Then, a process of an old component test is deactivated and detached by means of names $r_{stopTest}$ and $r_{detachTest}$. A process, which describes behaviour of the new component (i.e. the actual test script), is attached via name $r_{teAttach}$ as a subcomponent, bound via names $p'_{bindTInt}$ and $p'_{bindTRes}$, activated via name $r'_{startTest}$, and finally, it is executed via name $p'_{executeWithID}$ with a new name id (the identifier). Processing of Ctr'_{core} continues after the identical id is received via name p_{cDone}, i.e. the test script is finished and its results forwarded outside.

Core behaviour of components environment and test depends on a specific implementation of the testing environment and on a specific test script. However, for demonstrating purposes, we define process abstractions Env_{core} and $Test_{core}$:

$$\begin{aligned}
Env_{core} &\overset{def}{=} (p_{eInteract}).Env'_{core}\lfloor p_{eInteract} \rfloor \\
Env'_{core} &\overset{\Delta}{=} (p_{eInteract}).p_{eInteract}(ret).((val)\overline{ret}\langle val \rangle \mid Env'_{core}\lfloor p_{eInteract} \rfloor) \\
Test_{core} &\overset{def}{=} (p_{executeWithID}, r_{tInteract}, r_{tResult}).p_{executeWithID}(id). \\
&\quad\ (ret)(\overline{r_{tInteract}}\langle ret \rangle.ret(val).\overline{r_{tResult}}\langle val, id \rangle)
\end{aligned}$$

Process constant Env'_{core} receives a request from a test script via name $p_{eInteract}$ and returns a new name val as a reply. Process abstraction $Test_{core}$ receives identifier id via name $p_{executeWithID}$, sends a request to a process representing behaviour of a test environment via name $r_{tInteract}$, receives a reply and forwards it as the test's results together with id via name $r_{tResult}$.

6.2 Behaviour of a Composite Component

To assemble (sub)components into a composite component, we need to implement control actions. Components, primitive or composite, provide control interfaces for referencing their provided functional interfaces, binding their required functional interfaces (to the referred provided interfaces), and controlling their life-cycle (to start and stop the components). Moreover, each composite component provides its subcomponents with (internal) control interfaces for attaching and detaching other subcomponents, exporting their functional interfaces as the composite component's (external) functional interfaces, and importing the composite component's (external) functional interfaces to its subcomponents.

Behaviour associated with those control actions can be described in π-calculus, however, full definitions of related process abstractions [7] are out of the scope of this paper. For purpose of the following description, let us assume that $Ctrl_{Ifs}\langle r_1, \ldots, r_n, p^s_1, \ldots, p^s_n, p_1, \ldots, p_m, p^g_1, \ldots, p^g_m\rangle$ represents behaviour, which is associated with binding of interfaces represented by names $r_1, \ldots, r_n$ via control interfaces represented by names $p^s_1, \ldots, p^s_n$ and referencing of interfaces represented by $p_1, \ldots, p_m$ via control interfaces represented by $p^g_1, \ldots, p^g_m$. Moreover, let us assume that $Ctrl_{EI}\langle r_1, \ldots, r_n, p_1, \ldots, p_m, r'_1, \ldots, r'_m, p'_1, \ldots, p'_n\rangle$ represents behaviour of interconnections between external required and provided interfaces represented by names $r_1, \ldots, r_n$ and $p_1, \ldots, p_m$ and internal provided and required interfaces represented by names $p'_1, \ldots, p'_n$ and $r'_1, \ldots, r'_m$, respectively. Finally, let us assume that $Ctrl_{SS}\langle s_0, s_1, a\rangle$ represents behaviour, which is associated with a component's life-cycle (s_0 for stopping and s_1 for starting the component) and attaching new subcomponents (via a). Let us define an auxiliary constant application $Wire\lfloor x, y, d\rfloor$, which can receive a message via name x (an input) and send it via name y (an output) repeatedly till it receives a message via name d (i.e. disable processing). Detailed definitions of the above mentioned process abstractions and constants can be found in [7].

Behaviour of components output, environment, and test including their control parts can be defined as process abstractions Out, Env, and $Test$, respectively:

$$\begin{aligned}
Out &\overset{def}{=} (s_0, s_1, p^g_{oResult}, p^s_{oDone}, p^s_{oReply}).(p_{oResult}, r_{oDone}, r_{oReply}) \\
&\quad (Ctrl_{Ifs}\langle p_{oResult}, p^g_{oResult}\rangle \mid Ctrl_{Ifs}\langle r_{oDone}, p^s_{oDone}\rangle \\
&\quad \mid Ctrl_{Ifs}\langle r_{oReply}, p^s_{oReply}\rangle \mid Out_{core}\langle p_{oResult}, r_{oDone}, r_{oReply}\rangle) \\
Env &\overset{def}{=} (s_0, s_1, p^g_{eInteract}).(p_{eInteract}) \\
&\quad (Ctrl_{Ifs}\langle p_{eInteract}, p^g_{eInteract}\rangle \mid Env_{core}\langle p_{eInteract}\rangle) \\
Test &\overset{def}{=} (s_0, s_1, p^g_{executeWithID}, p^s_{tInteract}, p^s_{tResult}). \\
&\quad (p_{executeWithID}, r_{tInteract}, r_{tResult})(Ctrl_{Ifs}\langle r_{tInteract}, p^s_{tInteract}\rangle \\
&\quad \mid Ctrl_{Ifs}\langle p_{executeWithID}, p^g_{executeWithID}\rangle \mid Ctrl_{Ifs}\langle r_{tResult}, p^s_{tResult}\rangle \\
&\quad \mid Test_{core}\langle p_{executeWithID}, r_{tInteract}, r_{tResult}\rangle)
\end{aligned}$$

Behaviour of component controller is defined as process abstraction Ctr with free names $r_{teAttach}$, $r_{detachTest}$, $r_{stopTest}$, $r_{provRefEInt}$ and $r_{provRefORes}$ representing required control interfaces of other components:

$$\begin{aligned}
Ctr &\overset{def}{=} (s_0, s_1, p^g_{cDone}, p^g_{teExecTest}, \\
&\qquad r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}). \\
&\quad (p_{cDone}, p_{teExecTest})(Ctrl_{Ifs}\langle p_{cDone}, p^g_{cDone}\rangle \\
&\quad \mid Ctrl_{Ifs}\langle p_{teExecTest}, p^g_{teExecTest}\rangle \mid Ctr_{core}\langle p_{cDone}, p_{teExecTest}, \\
&\qquad r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}\rangle)
\end{aligned}$$

Behaviour of composite component testEnvironemt, i.e. the implementation of the core of service `TestEnvironment`, is described as process abstraction TE_{comp}:

$$
\begin{aligned}
TE_{comp} \overset{def}{=}\ & (s_0, s_1, p^g_{executeTest}, p^s_{asyncRepltET}).(p_{executeTest}, r_{teExecTest},\\
& p^s_{teExecTest}, r_{asyncRepltET}, p_{teReply}, p^g_{teReply}, p_{teAttach})\\
& (Ctrl_{Ifs}\langle p_{executeTest}, p^g_{executeTest}\rangle \mid Ctrl_{Ifs}\langle r_{teExecTest}, p^s_{teExecTest}\rangle\\
& \mid Ctrl_{Ifs}\langle r_{asyncRepltET}, p^s_{asyncRepltET}\rangle \mid Ctrl_{Ifs}\langle p_{teReply}, p^g_{teReply}\rangle\\
& \mid Ctrl_{EI}\langle p_{executeTest}, r_{teExecTest}\rangle \mid Ctrl_{EI}\langle p_{teReply}, r_{asyncRepltET}\rangle\\
& \mid Ctrl_{SS}\langle s_0, s_1, p_{teAttach}\rangle \mid TE'_{comp}\langle p_{teAttach}, p^s_{teExecTest}, p^g_{teReply}\rangle)\\
TE'_{comp} \overset{def}{=}\ & (p_{teAttach}, p^s_{teExecTest}, p^g_{teReply}).(s^{ctr}_0, s^{ctr}_1, s^{out}_0, s^{out}_1, s^{env}_0, s^{env}_1,\\
& p^g_{cDone}, p^g_{eInteract}, p^g_{oResult}, p^g_{teExecTest}, p^s_{oDone}, p^s_{oReply},\\
& r_{detachTest}, r_{provRefEInt}, r_{provRefORes}, r_{stopTest}, r_{teAttach})\\
& (Ctr\langle s^{ctr}_0, s^{ctr}_1, p^g_{cDone}, p^g_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest},\\
& r_{provRefEInt}, r_{provRefORes}\rangle \mid Env\langle s^{env}_0, s^{env}_1, p^g_{eInteract}\rangle\\
& \mid Out\langle s^{out}_0, s^{out}_1, p^g_{oResult}, p^s_{oDone}, p^s_{oReply}\rangle \mid (d)\overline{p_{teAttach}}\langle s^{ctr}_0, s^{ctr}_1, d\rangle\\
& \mid (d)\overline{p_{teAttach}}\langle s^{out}_0, s^{out}_1, d\rangle \mid (d)\overline{p_{teAttach}}\langle s^{env}_0, s^{env}_1, d\rangle\\
& \mid Test_{plug}\langle r_{detachTest}, r_{stopTest}\rangle \mid (d)Wire\lfloor r_{provRefEInt}, p^g_{eInteract}, d\rfloor\\
& \mid (d)Wire\lfloor r_{provRefORes}, p^g_{oResult}, d\rfloor \mid (d)Wire\lfloor r_{teAttach}, p_{teAttach}, d\rfloor\\
& \mid (ret)(\overline{p^g_{teExecTest}}\langle ret\rangle.ret(p_{teExecTest}).\overline{p^s_{teExecTest}}\langle p_{teExecTest}\rangle)\\
& \mid (ret)(\overline{p^g_{teReply}}\langle ret\rangle.ret(p_{teReply}).\overline{p^s_{oReply}}\langle p_{teReply}\rangle)\\
& \mid (ret)(\overline{p^g_{cDone}}\langle ret\rangle.ret(p_{cDone}).\overline{p^s_{oDone}}\langle p_{cDone}\rangle)\ )\\
Test_{plug} \overset{def}{=}\ & (r_{detachTest}, r_{stopTest}).(r_{detachTest} \mid r_{stopTest})
\end{aligned}
$$

Process abstraction TE'_{comp}, which is applied in process abstraction TE_{comp}, creates concurrent processes given by pseudo-applications of Ctr, Out, and Env and sends their names $s^{\cdot\cdot}_0$ and $s^{\cdot\cdot}_1$ via name $p_{teAttach}$, i.e. attaches components controller, output, and environment, respectively, as subcomponents of component testEnvironment. It also interconnects names representing required and provided control interfaces of the components by means of three constant applications of $Wire$. Concurrently with the previous step, TE'_{comp} applies process abstraction $Test_{plug}$ and binds name $p_{teExecTest}$ of the pseudo-application of Ctr to name $r_{teExecTest}$ of the pseudo-application of TE_{comp}, name p_{cDone} of Ctr to name r_{cDone} of Out, and name $p_{teReply}$ of TE_{comp} to name $r_{teReply}$ of Out. The pseudo-application of process abstraction $Test_{plug}$ handles requests initiated by the pseudo-application of Ctr and received by names $r_{stopTest}$ and $r_{detachTest}$ to stop and to detach a process representing behaviour of a previous but non-existent component with a test script (e.g. a non-existent predecessor of component test).

7 Related Work and Discussion

Related works relevant to our subject can be divided into two groups, as formal approaches to describe service-oriented architectures (SOAs) and as formal approaches to describe component-based systems (CBSs). In this section, we outline current state of the art in both groups and discuss advantages and drawbacks of our approach, which intends to bridge the gap and to provide formal description of service-oriented architecture from choreography of services to individual components of underlaying component-based systems.

In the first group, there are approaches mostly based on *Business Process Execution Language for Web Services* [8], such as [9], [10] or [11]. Those approaches focus on the web services, as a specific implementation of SOA, and provide formal description of choreography and orchestration based on business processes. The description ends up at the level of individual services implementing business processes and does not include underlying CBSs.

The second group consists of several component models[3] [12], such as Darwin/Tracta [13], Fractal [14] or SOFA 2.0 [15]. Those models usually focus only on pure CBSs without considering SOA at the higher level of abstraction. In some cases [16], the component models brings features of SOA into CBD, so that SOA becomes a specific case of a CBS. However, this solution mixes two different levels of abstraction (see Section 1).

Our approach is similar to the Reo coordination language [17], which is also based on π-calculus and able to describe both service in SOA and components in CBSs. In comparison with Reo and the above mentioned approaches (especially those in the second group), our approach describes services and components separately and with respect to their differences (i.e. services are not components and vice versa). We allow to go smoothly from services level to components level and describe behaviour of a whole system, services and components, as one π-calculus process. Moreover, we use standard polyadic π-calculus without any special extensions, which allows to utilise a wide range of existing tools for model-checking of π-calculus processes and formal verification of their properties.

However, our approach can have also drawbacks, e.g. complex description of behaviour of primitive components' control actions processing or insufficient visibility of a component-based system's structure during its evolution. After several dynamic reconfigurations and a corresponding sequence of reductions of the π-calculus process, it may be difficult to determine a final configuration from the resulting π-calculus process, especially without knowledge of the exact sequence of reductions.

8 Conclusion and Future Work

We have demonstrated an approach to formal description of behaviour of service-oriented architecture on a case study of a testing environment of a railway interlocking control system. The approach is innovative, it captures behaviour of services as well as behaviour of underlying systems of components, yet it distinguishes these two levels. Future work is related to integration of the approach into modelling tools and automatic generation of the formal description.

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

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

References

1. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (August 2005)
2. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Journal of Information and Computation **100** (September 1992) 41–77
3. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. New Ed edn. Cambridge University Press (October 2003)
4. Donini, R., Marrone, S., Mazzocca, N., Orazzo, A., Papa, D., Venticinque, S.: Testing complex safety-critical systems in SOA context. In: CISIS, Los Alamitos, CA, USA, IEEE Computer Society (December 2008) 87–93
5. 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 (2008)
6. Bowen, J.P., Stavridou, V.: Safety-critical systems, formal methods and standards. IEE/BCS Software Engineering Journal **8**(4) (July 1993) 189–209
7. Rychlý, M.: A component model with support of mobile architectures and formal description. e-Informatica Software Engineering Journal **1**(2) (2009) In print.
8. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for Web Services, v1.1. Technical report, IBM (2003)
9. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets
10. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming **70**(1) (January 2007) 96–118
11. Weidlich, M., Decker, G., Weske, M.: Efficient analysis of BPEL 2.0 processes using π-calculus. In: APSCC '07: Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference, Washington, DC, USA, IEEE Computer Society (2007) 266–274
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. Giannakopoulou, D.: Model Checking for Concurrent Software Architectures. PhD thesis, Imperial College of Science, Technology and Medicine University of London, Department of Computing (January 1999)
14. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium (February 2004)
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. 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
17. Diakov, N.K., Arbab, F.: Compositional construction of Web Services using Reo. In Bevinakoppa, S., Hu, J., eds.: Proc. of International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI 2004), INSTICC Press (April 2004) 49–58