# Formal-Based Component Model with Support of Mobile Architecture

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
In this article, an approach to modelling of component-based systems and formal description of their behaviour is proposed. It is based on a novel component model defined by a metamodel in a logical view and by description in the $\pi$-calculus in a process view. The model addresses dynamic aspects of software architectures including component mobility. Furthermore, a method of behavioural modelling of service-oriented architectures is proposed to pass smoothly from service to component level and to describe behaviour of a whole system as a single $\pi$-calculus process. The support of dynamic architecture and the integration with service-oriented architecture compromise the main advantages of the approach.

## Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Languages; H.3.4 [**Systems and Software**]: Distributed systems; D.2.4 [**Software/Program Verification**]: Formal methods

## Keywords
Component-based development, Service-oriented architecture, Component model, Formal specification

## 1. Introduction
Globalisation of information society and its progression create needs for extensive and reliable information technology solutions. Several new requirements on information systems have emerged and significantly affected software architectures of these systems. The current information systems can not be realised as monoliths, but tend to be distributed into networks of quite autonomous, but cooperative, components communicating asynchronously via messages of appropriate formats. Loose binding between those components allows to establish and destroy their interconnections dynamically at runtime, on demand, and according to various aspects (e.g. quality of services provided or required by the components); to clone the components and to move them into different contexts (known as "component mobility"); to create, destroy and update the components dynamically at runtime; etc.

The dynamic aspects of software architectures and the component mobility bring new problems in the domain of software engineering, as it is described in Sec. 3.1. The component-based systems (CBSs) are getting involved, and a formal specification of evolution of their architectures is necessary, particularly in critical applications. The current problems and the state of the art, which is summarised in Sec. 2, provide us with adequate motivation and form objectives of the research in Sec. 3.2.

To provide a method for modelling of CBSs and formal description of their behaviour, we propose an approach that is based on a novel component model defined by a metamodel in a logical view in Sec. 4.2 and by description in the $\pi$-calculus in a process view in Sec. 4.3. We show that the component model addresses the dynamic aspects of software architectures including the component mobility. Furthermore, in Sec. 4.4, we propose a method of behavioural modelling of service-oriented architectures (SOAs) to pass smoothly from service level to component level and to describe behaviour of a whole system, services and components, as a single $\pi$-calculus process.

## 2. State of the Art
This section deals with software architecture and component-based development. It describes the state of the art of component models and architecture description languages for dynamic and mobile architectures.

### 2.1 Software Architecture and Component-Based Development
The **software architecture** as "the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution" [15] can be described using *logical (structural)* and *process (behavioural) view* [16]. The first describes logical structure

---

*Recommended by thesis supervisor:
Assoc. Prof. Jaroslav Zendulka
Defended at Faculty of Information Technology, Brno University of Technology on February 19, 2010.

© Copyright 2010. All rights reserved. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from STU Press, Vazovova 5, 811 07 Bratislava, Slovakia.

Rychlý, M. Formal-based Component Model with Support of Mobile Architecture. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 2, No. 1 (2010) 13-25

of the system, while the second describes concurrency and synchronisation aspects of the system, e.g. behaviour of components, evolution of the architecture in time, etc.

We can distinguish three types of software architectures according to their evolution in dependence on changes of their environment [27]: static architecture, dynamic architecture, and mobile (fully dynamic) architecture. In the *static architecture*, after initialisation of a system, new connections between the system's components can not be created and existing connections can not be destroyed. In the *dynamic architecture*, there exist rules of evolution of a software system in time (also called a "dynamics"), i.e. its components and connections are created and destroyed during the system's runtime according to the rules from its design-time. Finally, the *mobile architecture* is a dynamic architecture of a system where components can change their context in the system's logical structure during its execution (i.e. "component mobility") according to rules from its design-time and functional requirements.

The **component-based development** (CBD, see [34]) is a software development methodology strongly oriented to composability and re-usability in a software system's architecture. In the CBD, from a structural point of view, a system is composed of *components*, 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*.

A static architecture has only one way how to connect components and connectors into a resulting system, i.e. there is only one configuration. Dynamic and mobile architectures enable software systems to change their architectures during their runtimes. It means runtime modifications of the configuration, i.e. *reconfiguration*. Description of the reconfiguration [27] includes description of specific *actions*, which are consumed and produced by a system (inputs, outputs, and internal actions); *relationships* between the actions, how the input actions are processed by the system; and *changes* of an architecture according to the actions, i.e. processes of creation and destruction of its components, connectors and reconfiguration.

In CBD, components can be primitive or composite. The *primitive components* are realised directly, beyond the scope of architecture description. The *composite components* are decomposable into systems of subcomponents at the lower level of architecture description. This composition forms a *component hierarchy*.

### 2.2 Component Models and Architecture Description Languages

**Component models** are specific metamodels of software architectures supporting the CBD. The component models should define syntax, semantics, and composition of components [17]. They are systems of rules for components, connectors, configurations, rules for changes according to the dynamic architecture (rules for reconfigurations), etc. Several component models has been proposed [18] including the models. Those models differ particularly in definitions of connectors (explicit or implicit definitions) and implementation of advanced features of dynamic or mobile architectures. In this section, we focus on component models with formal bases.

*Component model Wright* [1] uses the process calculus of *Communicating Sequential Processes* (CSP). The component model defines a *component* as a structure composed of two parts, an interface and a "component-spec". The *interface* consists of a finite number of required and provided *ports* and corresponding CSP processes for specific input and output events, respectively. The *component-spec* defines interactions between the ports as a composition of their CSP processes. Interactions of components are defined by *connectors* and described by specific CSP processes. Finally, a *configuration* describes actual bindings of the components and the connectors. Limitations of Wright are given by the used formalism (the descriptions in CSP support only systems with static architectures).

*Component model Darwin* [19, 12] allows distributed systems to be hierarchically composed of sets of component instances and their interconnections at each level of the hierarchy. Each *component* is defined by its required and provided services (interfaces), which allow it to interact with other components. *Composite components* are defined by declaring instances of internal components and "required-provided" bindings between those components. Services of the internal components that cannot be satisfied can be declared as visible at a higher level of the hierarchy, as the services of the composite components. Darwin [19] allows to describe behaviour of components in the calculus of mobile processes (the $\pi$-calculus). It supports a subset of *dynamic architectures*, which permits dynamic instantiation of new components at runtime, but does not allow specification of dynamic bindings or component removal. Later, *Tracta* approach [12] allows to describe behaviour of components as *Labelled Transition Systems* (LTSs) with an algebra of *Finite State Processes* (FSP) as a specification language.

In *component model SOFA* [29], 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. A primitive component has a primitive architecture and it is directly implemented by a software system. The *architecture* of a composite component is a "grey-box" implementation view, which describes its direct subcomponents and their interconnections via interfaces: *binding* of required to provided interfaces, *delegating* of a component's provided interfaces to provided interfaces of the component's subcomponent, *subsuming* of required interfaces of a component's subcomponent to the component's required interfaces, and *exempting* of subcomponent's interfaces from any connection. SOFA uses a *Component Definition Language* (CDL, [22]), which extends features of OMG IDL to allow specification of software components. Behaviour of a component (its interface, frame, and architecture) is formally described by means of *behaviour protocols* [37]. The protocols support static architecture, but allow to describe *dynamic update* of an architecture of a component during a system's runtime.

*Component model SOFA 2.0* [8], as a new version of component model SOFA, aims at removing several limitations of the original version, mainly the lack of support of dynamic reconfigurations, well-structured and extensible control parts of components, and multiple styles of

communication between components. It permits dynamic reconfigurations predefined at design-time by *reconfiguration patterns*: nested factory, component removal, and utility interface. The utility interfaces can be freely passed among components and bound independently of their components' levels in architecture hierarchy (then, SOA becomes a specific case of a component model where all components (services) are interconnected solely via their utility interfaces).

*Component model Fractal* [6, 7] forms a *component* 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* of the enclosing component, which acts as a composition operator. A component can interact with its environment through *operations* at *external interfaces* of the component's controller, while *internal interfaces* are accessible only from the component's subcomponents. A *functional interface* requires or provides functionalities of a component, while a *control interface* is a server interface, which provides operations for *introspection* of the component and to control the component's *configuration*, namely attribute, binding, content, and life-cycle control. Operations on functional interface can not fire control operations. Behaviour of Fractal components can be formally described by means of *parametrised networks of communicating automata* language [5]. Behaviour of each primitive component is modelled as a finite state *parametrised labelled transition system* (pLTS), while behaviour of a composite component is defined using a *parametrised synchronisation network* (pNet): a set of global parametrised actions and a transducer, which is a synchronisation product of the subcomponents' pLTSs. Changes of the transducer's state represent possible reconfigurations of the composite component's architecture.

**Architecture description languages** (ADLs, see [35]) are languages for describing software systems' architectures, which focus on high-level structures of overall applications rather than implementation details of any specific source modules. The ADLs can be parts of component models, where they are used for description of a software system's logical structure (the logical view, see Sec. 2.1). Several ADLs have been proposed [21]: for modelling of software architectures within a particular domain, as general-purpose architecture modelling languages, with and without component models and formal bases, etc. In the rest of this section, we aim at the ADLs that do not depend directly on component models.

*Language* ACME [11] has been developed in order establish a common basis for the ADLs and to enable integration of their support tools. It defines components and connectors, systems (as configurations of components and connectors), ports (as interfaces of a component), roles (of interfaces of a connector, which they act in communication), representations (hierarchical decompositions of components and connectors), and rep-maps (mappings between a composite component's or connector's internal architecture and its external interface). Other aspects of architectural description can be represented with *property lists*. The ACME does not provide any certain semantic model. The property lists, structural constraints, etc. must be described in terms of other ADLs' semantic model. Therefore, the ACME itself is not suitable for description of a system's software architecture and should be used only in association with other ADL (where ACME acts as the ADL's exchange language).

*Unified Modelling Language* (UML, see [26]) provides three possible strategies for modelling of software architectures [20]: to use UML "as is", to constrain its metamodel using UML's built-in extension mechanisms, and to extend the metamodel by new architectural concepts. Each approach has potential advantages and disadvantages (e.g. limited semantics of standard UML or incompatiblity of the extensions with UML-compliant tools). The UML 2 has introduced description of hierarchical architecture of CBSs [3] by means of *structured classes*, i.e. the classes that allow nesting of other classes. Its specification [26] states that "a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment". The components are drawn as specific classes stereotyped «`component`» interconnected by means of "assembly connectors" binding their interfaces in a "lollipop" style notation. Moreover, the components can be used to represent many different entities, which are distinguished by several stereotypes.

*ArchWare ADL* (see [4]) provides a core (runtime) structure and behaviour constructs to describe dynamic software architectures. It is a formal specification language designed to be executable (by a virtual machine) and to support automated verification. The ArchWare ADL is founded on three formal models: $\pi$-ADL [27] that contains the core structure and behaviour constructs with the higher-order typed $\pi$-calculus as a formal basis; $\sigma\pi$-ADL [4] that contains style constructs for defining a base component-connector style and other derived styles; and $\mu\pi$-AAL that is extension of the modal $\mu$-calculus for description of behavioural and structural properties of communicating and mobile architectural elements. The behaviour constructs from the base language copy the $\pi$-calculus constructs.

## 3. Motivation

This section aims at clarifying the motivation of the research. A problem is stated in terms of the state of the art from Sec. 2 and objectives of the research are established. The detailed description of the research objectives can be found in the author's dissertation [31].

### 3.1 Statement of the Problem

The current component models and ADLs have many shortcomings in support of mobile architectures, incorporation of component-based design into SOA and into software development processes in general. Their formal bases or models usually do not consider component mobility (e.g. pNets in Fractal [5], behaviour protocols in SOFA [37], and reconfiguration patterns in SOFA 2.0 [14]), prefer strict isolation of functional parts of components from their controllers (functional operations can not fire control operations; e.g. restrictions of pNets in a formal description of Fractal components [5]), do not support description of SOAs where individual services can be implemented as underlying CBSs (e.g. in Fractal component model [7] or in the ArchWare project [4]), etc.

Moreover, the component models do not describe related methods of CBD and their incorporation into well-established software development processes of standard software systems [13]. Modelling of the CBSs during

the development processes brings many issues (e.g. as a consequence of different conceptions of components in the component models [17], in component diagrams of UML [26] or software component architectures [28]).

### 3.2 Objectives of the Research

The statement of the problems of current component models and corresponding ADLs provides us with adequate motivation. A *general objective of the research* is to design a component model for mobile architectures. The model has to provide suitable formal basis and should be applicable to modelling of CBSs as well as SOAs.

*Specific objectives of the research* that fulfil the general objective include: the development of the component model and its formal basis supporting features of mobile architectures and addressing the current issues of CBD; integration of functional operations and relevant behaviour of components with control operations enabling dynamic reconfiguration; a method of application of the component model in SOAs with mapping rules between services and CBSs described by means of the component model; description of a supporting environment that allows integration of the component model and utilisation of its formal basis into software development processes; and finally, demonstration of an application of the proposed approach on a case study and evaluation of its effectiveness and robustness over the existing conventional approaches.

## 4. Methods of Realisation

In order to reach the stated objectives, we propose a high-level component model addressing the current issues of CBD. It allows dynamic reconfiguration, component mobility, and combination of control and business logic of components. Behavioural description of individual components and their mutual communication in CBSs is based on the $\pi$-calculus.

The component model can be presented in two views: logical (structural) view and process (behavioural) view. At first, we introduce the component model's metamodel, which describes basic entities of the component model and their relations. The second view, is focused on behaviour of the component model's entities, especially on component mobility. Finally, we describe behaviour of services in SOAs and their underlying implementations as CBSs.

### 4.1 Formal Base

To describe behaviour of components in CBSs and services in SOAs in formal way, we use the $\pi$-calculus, known also as a *calculus of mobile processes* [23]. 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 $\pi$-calculus, while the *names* are anything else, e.g. communication links (known as "ports"), variables, constants, 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 (e.g. as communication links). We suppose basic knowledge of the fundamentals of the $\pi$-calculus, a theory of mobile processes, according to [33]:

- $\overline{x}\langle y\rangle.P$ 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 of name $z$ in process $P$ (the scope may change as a result of interaction between processes);
- $(\tilde{x})P = (x_1, x_2, \ldots, x_n)P = (x_1)(x_2)\ldots(x_n)P$ 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 $\pi$-calculus processes can be parametrised as an *abstraction*, 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 $x$. The abstractions can be used in 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 $K \triangleq (x).P$. The result of the reduction yields process $P\{y/x\}$.

### 4.2 Component Model: Logical View

The component model for mobile architectures is described as a metamodel in the context of a four-layer modelling architecture. The metamodel is implemented in OMG's *Meta Object Facility* (MOF, [24]), which is used as a meta-metamodel. The modelling architecture comprises the following four layers:

**M0** An information layer, which is comprised of the actual data objects. It contains particular instances of CBSs, their runtime configurations, specific deployments of their components and connectors, etc.

**M1** A model layer, which contains models of the M0 data. The models include structure and behaviour models that describe different perspectives of CBSs such as, for example, UML component models.

**M2** A metamodel layer provides a language that can be used to build M1 models. Component models fall in this layer, as well as models of the UML language.

**M3** A meta-metamodel layer, which is used to define modelling languages. It holds a model of the information from M2, e.g. MOF.

In the four-layer modelling architecture, the models in lower layers use classes from metamodels in upper layers to create their objects. In the context of CBD, a specific CBS (layer M0) contains instances of elements from its model (layer M1). The model contains instances from a specific component model (a metamodel in layer M2), which is described by a given meta-metamodel (layer M3).

#### 4.2.1 Metamodel

This section deals with description of the component model for mobile architectures as a metamodel. The metamodel is defined in *Meta Object Facility* version 2.0 (MOF, [24]). MOF is in layer M3 in the four-layer modelling architecture (see Sec. 4.2). It is defined in two parts: *Essential MOF* and *Complete MOF* (EMOF and CMOF). The EMOF contains packages `Basic`, `Reflection`, `Identifiers`, and `Extension`, which form a minimal set of modelling elements to define simple metamodels. The CMOF extends EMOF by `Constructs` package from UML 2 Core (see [25]). For purposes of this article, the EMOF is sufficient to describe the component model.

The component model, as a model of layer M2 in the four-layer modelling architecture, can be described by means of UML 2 diagrams in two contexts: as an *object diagram* of instances of EMOF classes from layer M3 (entities in layer M2 are instances of classes in layer M3, i.e. it is described as "a model") and as a *class diagram* from layer M1 (entities in layer M1 are instances of classes in layer M2, i.e. it is described as "a metamodel").

For better clearness, the component model will be described as an UML 2 class diagram from layer M1. To reuse well-established concepts of MOF, the component model's metamodel extends EMOF classes `EMOF::NamedElement`, `EMOF::TypedElement`, and `EMOF::Operation`, which are outlined in Fig. 1. A complete definition of the EMOF classes can be found in [24].

**Components and Interfaces.** Fig. 2 describes the first part of the component model as an extension of EMOF. The metamodel defines an abstract component, its realisations as a primitive component and a composite component, and their interfaces. All classes of the metamodel inherits (directly or indirectly) from class `EMOF::NamedElement` in package `Basic` of EMOF.

In our approach, a *component* as an active communicating entity of a CBS can be described according to two views: as an abstract component without considering its internal structure (in a "black-box" view) and as a component realisation in the form of a primitive component or a composite component (in a "grey-box" view). The *abstract component* (class `Component` in the metamodel) can communicate with neighbouring components via its interfaces (class `Interface`). The interfaces can be provided (class `ExternalProvInterface`) or required (class `ExternalReqInterface`) by the component.

The component realisation can be primitive or composite. The *primitive component realisation* (class `PrimitiveComponent`) is implemented directly, beyond the scope of architecture description. It is a "black-box" with described observable behaviour (attribute `behaviouralDescription`). The *composite component realisation* (class `CompositeComponent`), as a "grey-box", is decomposable on a system of subcomponents at the lower level of architecture description. Those subcomponents are represented by abstract components (class `Component` and relation "consists of"). Moreover, every composite component realisation can communicate with its subcomponents via its provided (class `InternalProvInterface`) and required (class `InternalReqInterface`) internal interfaces (relations "provides inside" and "requires inside", respectively).

The specific interfaces have to implement methods `getOwner()`, which return their owners, i.e. objects that act as the abstract components in a case of the abstract component interfaces or as instances of the composite component realisations in a case of their internal interfaces (see `owner` roles in the relations in Fig. 2).

**Composite Components and Binding.** Binding is a connection of required and provided interfaces of the identical types into a reliable communication link. It is described in Fig. 3. Interfaces of a component (classes `ExternalProvInterface` and `ExternalReqInterface`) can be provided to and required from its neighbouring components, while interfaces of a composite component realisation (classes `InternalProvInterface` and `InternalReqInterface`) can be provided to and required from its subcomponents only. Therefore, we distinguish three types of the binding (the realisations of class `Binding`):

1. Binding *of provided interfaces to required interfaces in the same composite component realisation* is represented by class `BindSiblings`. The interfaces have to be internal interfaces of the composite component realisation or external interfaces of subcomponents in the same composite component realisation. The binding interconnects required interfaces (class `RequiredInterface`) via relations "bound from" to provided interfaces (class `ProvidedInterfaces`) via relations "bound to".
2. Binding *of external provided interfaces* of a composite component realisation *to its internal required interfaces* is represented by class `BindInward`. The external interfaces are provided to neighbouring components of the composite component acting as an abstract component (relation "imports from" an instance of class `ExternalProvInterface`), while the internal interfaces are required from the composite component's subcomponents (relation "exports to" an instance of class `ExternalReqInterfaces`).
3. Binding *of internal provided interfaces* of a composite component realisation *to its external required interfaces* is represented by class `BindOutward`. The internal interfaces are provided to the composite component's subcomponents (relation "exports from" an instance of class `InternalProvInterface`), while the external interfaces are required from neighbouring components of the composite component acting as an abstract component (relation "exports to" an instance of `ExternalReqInterfaces`).

The bindings (i.e. instances of the realisations of class `Binding`) are owned by the composite component realisa-

**Figure 1: A simplified part of the EMOF [24] with classes that will be extended in the metamodel.**

**Figure 2: Abstract component, realisations, and interfaces in the metamodel.**

**Figure 3: Binding and its different realisations between interfaces of a composite component realisation in the metamodel. Classes `CompositeComponent` and `...Interface` are identical to the classes in Fig. 2.**

tions. Each binding can have a type (class `TypeOfBinding`), which can describe a communication style (buffered and unbuffered connection), a type of synchronisation (blocking and output non-blocking), etc.

**Types of the Interfaces.** To ensure type compatibility of interfaces in a binding, each interfaces has a type (class `TypeOfInterface`, which is a specialisation of class `EMOF::NamedElement` in package `Basic` of EMOF). Hierarchy of the types of interfaces is described in Fig. 4.

According to a scope of visibility of the interfaces in a composite component realisation, we can distinguish public, private, and protected interfaces. The *public interfaces* (classes realising `PublicIntType`) of a component can be accessed by its neighbouring components (via binding `BindSiblings`). If the component is a composite component realisation, its external public interfaces can be also accessed by its subcomponents and its internal public interfaces can be accessed by its neighbouring components (i.e. the interfaces can pass the component's border via its bindings `BindInward` and `BindOutward`). They can be interconnected by means of all kinds of bindings.

Contrary to the public interfaces, the *private interfaces* (classes realising `PrivateIntType`) are specific types of interfaces, which can be provided only by a composite component realisation and only to its subcomponents as the component's internal interfaces[1]. They can be interconnected only by means of binding `BindSiblings`.

Finally, the *protected interfaces* (classes realising `ProtectedIntType`) of a component can be accessed by its neighbouring components as the component's external interfaces (if the component is a composite component realisation, they are not reachable by its subcomponents). They can be interconnected only via `BindSiblings`.

According to functionality, we can distinguish the following types of interfaces[2] (see Fig. 4):

- Public interface `Operation`, which extends class `EMOF::Operation` from package `Basic` of EMOF and represents a business oriented service with typed input and output parameters.
- Protected interface `CtrlRefProvInterface` provides references to given provided interface `ProvidedInterface` of type `Operation`, while protected interface `CtrlBindReqInterface` allows to establish a new binding of specific required interface `RequiredInterface` of type `Operation` to a provided interface of another component formerly referred by means of `CtrlRefProvInterface`.
- Protected interfaces `CtrlStart` and `CtrlStop` allow to control behaviour of a component (i.e. to start and to stop the component, respectively).
- Private interfaces `CtrlAttach` and `CtrlDetach` provided by a composite component realisation allow to attach a new component as a subcomponent of the composite component realisation ("nesting" of the component) and detach an old subcomponent from the composite component realisation, respectively.
- Protected interface `CtrlClone` provides references of a fresh copy of a component.
- Protected interface `RefToInterface` is able to pass references of provided interfaces `ProvidedInterface` of type `Operation`, while public interface `RefToComponent` allows to pass references of a whole `Component` (required for component mobility).

[1]The private interfaces can be required by the subcomponents as their external interfaces, but they can not pass borders of the subcomponents (nor any other component). Here, the subcomponents must be primitive components.
[2]Type `Operation` denotes *functional interfaces*, while the others denote *control interfaces*.

### 4.3 Component Model: Process View

In this section, the component model is presented in the process view. Behaviour of individual components and their mutual communication is described by means of the $\pi$-calculus (see Sec. 4.1).

According to the metamodel from Sec. 4.2.1, each component of a CBS can be realised either as a primitive component or as a composite component. Since the *primitive component* is described as a "black-box", its behaviour has to be defined directly by its developer and can be described as a $\pi$-calculus process (a value of attribute `behaviouralDescription` in an instance of class `PrimitiveComponent`, see Fig. 2 in Sec. 4.2.1). The $\pi$-calculus process describes processing of names that represent the component's provided and required functional interfaces and names for specific control actions provided by the component via its control interfaces (e.g. requests to start or stop the component).

On the contrary to the primitive component, the *composite component* is decomposable at a lower level of component hierarchy into a system of subcomponents communicating via their interfaces and their bindings (i.e. a CBS; the component is a "grey-box"). Formal description of the composite component's behaviour is a $\pi$-calculus process, which is composition of processes representing behaviour of the component's subcomponents, processes implementing bindings between interfaces of the subcomponents (class `BindSiblings` in the metamodel), bindings of internal interfaces of the component to its external interfaces (classes `BindInward` and `BindOutward`), and processes describing specific control actions of the component's control interfaces (e.g. requests to start or stop the composite component including their distribution to the component's subcomponents, etc.).

#### 4.3.1 Notation

Before $\pi$-calculus processes describing behaviour of a component will be presented, we need to define the *component's interfaces* within the terms of the $\pi$-calculus, i.e. as names used by the $\pi$-calculus processes. In an external view of a component, i.e. for description of an abstract component (as a specific instance of class `Component`), we will use names $s_0, s_1, c, p_1^s, \ldots, p_n^s, p_1^g, \ldots, p_m^g$. In an internal view of a component, i.e. for description of a composite component (an instance of class `CompositeComponent`), we will use names $a,\ p_1'^s, \ldots, p_m'^s,\ p_1'^g, \ldots, p_n'^g$. In both cases, $n$ and $m$ are numbers of the component's required and provided functional interfaces, respectively (i.e. the component's external interfaces of type `Operation`), and the individual names have the following semantics:

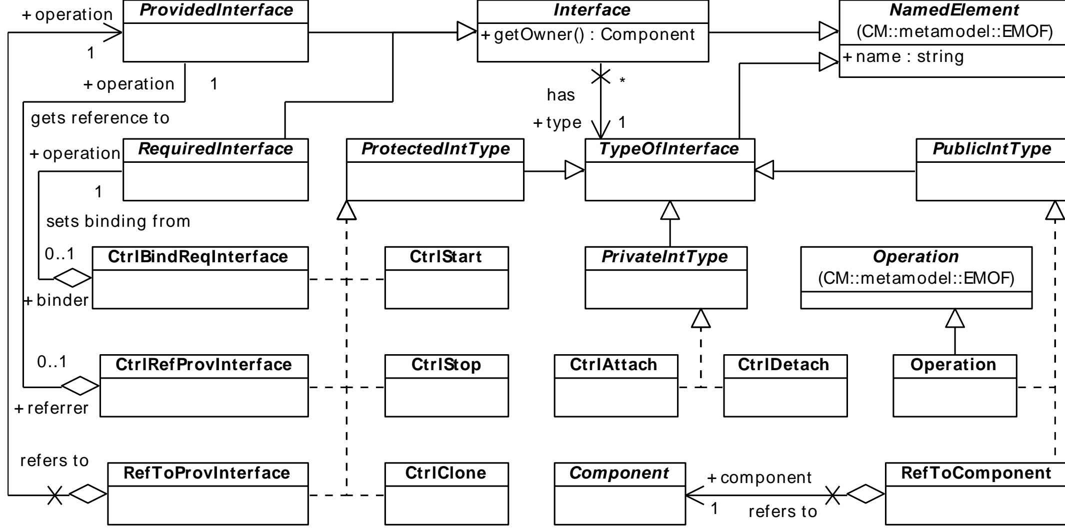


**Figure 4: Types of interfaces with class `Operation` extending `EMOF::Operation` in the metamodel. Classes `Interface`, `ProvidedInterface`, `RequiredInterface`, and `Component` are identical to the classes in Fig. 2.**

**via** $s_0$ – a running component accepts a request for its stopping[3] (an interface of type `CtrlStop`),

**via** $s_1$ – a stopped component accepts a request for its starting[3] (an interface of type `CtrlStart`),

**via** $c$ – a component accepts a request for its cloning and returns a new stopped instance of the component as a reply (an interface of type `CtrlClone`),

**via** $p_i^s$ – a component accepts a request for binding a specific provided functional interface (included in the request) to required functional interface $r_i$ (an interface of type `CtrlBindReqInterface`),

**via** $p_j^g$ – a component accepts a request for referencing provided functional interface $p_j$, which reference is returned as a reply (an interface of type `CtrlRef-ProvInterface` in the metamodel),

**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 requests to detach the subcomponent (the names represent interfaces of types `CtrlAttach` and `CtrlDetach`).

We should remark that there exists a relationship between names representing functional interfaces in the external view and names representing corresponding functional interfaces in the internal view of a composite component. The composite component interconnects its external functional interfaces $r_1, \ldots, r_n$ (required) and $p_1, \ldots, p_m$ (provided) accessible via names $p_1^s, \ldots, p_n^s$ and $p_1^g, \ldots, p_m^g$, respectively, to internal functional interfaces $p'_1, \ldots, p'_n$ (provided) and $r'_1, \ldots, r'_m$ (required) accessible via names $p_1'^g, \ldots, p_n'^g$ and $p_1'^s, \ldots, p_m'^s$, respectively.

As a result, requests received via external functional provided interface $p_j$ are forwarded to an interface that is bound to internal functional required interface $r'_j$ (and analogously for interfaces $p'_i$ and $r_i$). This ensures binding of external interfaces of the composite component to its internal interfaces and vice versa, as it has been described in the medamodel (see classes `BindInward` and `BindOut-ward` in Fig. 3 in Sec. 4.2.1).

### 4.3.2 Interface's References and Binding

At first, we define an auxiliary process $Wire$[4], 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 components' functional interfaces is done via their control interfaces. These control interfaces allow to get a reference to a component's functional provided interface (via an interface of type `CtrlRefProvInterface` in the metamodel) and use the reference to bind a functional required interface of another component (via an interface of type `CtrlBindReqInterface` in the metamodel). Process $Ctrl_{Ifs}$ describes processing of requests via the control interfaces as follows:

$$SetIf \triangleq (r, s, d).s(p).(\overline{d}.Wire\lfloor r, p, d\rfloor \mid SetIf\lfloor r, s, d\rfloor)$$

$$GetIf \stackrel{def}{=} (p, g).g(r).\overline{r}\langle p\rangle$$

$$Plug \stackrel{def}{=} (d).d$$

$$Ctrl_{Ifs} \stackrel{def}{=} (r_1, \ldots, r_n, p_1^s, \ldots, p_n^s, p_1, \ldots, p_m, p_1^g, \ldots, p_m^g). (\textstyle\prod_{i=1}^{n}(r_i^d)(Plug\langle r_i^d\rangle \mid SetIf\lfloor r_i, p_i^s, r_i^d\rfloor) \mid \textstyle\prod_{j=1}^{m}!GetIf\langle p_j, p_j^g\rangle)$$

where names $r_1, \ldots, r_n$, $p_1^s, \ldots, p_n^s$, $p_1, \ldots, p_m$, $p_1^g, \ldots, p_m^g$ have been defined in Sec. 4.3.1. Let us assume $Ctrl_{Ifs}$ shares its names $r_1, \ldots, r_n$ and $p_1, \ldots, p_m$ with a process describing a component's core functionality via its required and provided interfaces, respectively. Pseudo-application $GetIf\langle p_j, p_j^g\rangle$ enables process $Ctrl_{Ifs}$ to receive a name $x$ via $p_j^g$ and to send $p_j$ via name $x$ as a reply (it provides a reference to an interface represented by $p_j$). Constant application $SetIf\lfloor r_i, p_i^s, r_i^d\rfloor$ enables

[3]In a composite component, the requests are distributed to all subcomponents of the component.

[4]The process will be used also in the following parts of Sec. 4.3.

process $Ctrl_{Ifs}$ to receive a name $x$ via $p_i^s$, 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 binding of $r_i$, a request is sent via $r_i^d$ (in case it is the first binding of $r_i$, i.e. there is no former binding, the request is accepted by pseudo-application $Plug\langle r_i^d\rangle$).

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 an interface that is bound to internal functional required interface $r'_j$ (and analogously for interfaces $p'_i$ and $r_i$). This behaviour is described in process $Ctrl_{EI}$:

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

### 4.3.3 Control of a Component's Life-cycle

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

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

where names $s_0$ and $s_1$ represent the component's interfaces that accept stop and start requests, respectively (i.e. interfaces of types `CtrlStop` and `StrlStart` in the metamodel), and name $a$ that can be used to attach stop and start interfaces of the component's new subcomponent (at one step, i.e. via an interface of type `CtrlAttach`).

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 message $m$ via $s_0$ or $s_1$. This message 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 a case the component is running or $p_1$ in a case the component is stopped. When all subcomponents have accepted message $m$, the process of starting or stopping the component is finished, which is announced via name $r$, and the component is ready to receive new requests to stop or start, respectively.

Pseudo-application $Attach\langle a,p_0,p_1\rangle$ enables $Ctrl_{SS}$ to receive a message via $a$, i.e. 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 constant applications of process $Wire$. Third name received via $a$, $c_d$, can be used later to detach the previously attached stop and start interfaces.

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

### 4.3.4 Cloning of Components and Updating of Subcomponents

Cloning of a component allows to create the component's fresh copy and to transport it into different location, i.e. for attaching as a subcomponent of anther component. Process $Ctrl_{clone}$ describes processing of requests for clonning of a component as follows:

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

where the pseudo-application of $Component$ 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 $k$ 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 control interfaces for binding functional requested interfaces and referencing functional provided interfaces of the new component, i.e. names $p_1^s,\ldots,p_n^s$ via $r$ and names $p_1^g,\ldots,p_m^g$ via $p$, respectively.

The fresh copy of a component can be used to replace a subcomponent of a composite component. The process of update[6], which describes replacing of the old subcomponent with a new one, is not a mandatory part of the composite component's behaviour and its implementation depends on particular configuration of the component (e.g. ability of the component to update its subcomponents, a context of the replaced subcomponent, presence of parts of the component that have to be stopped during the update, etc.). For example, we can describe replacing a subcomponent as process $Update$:

$$\begin{aligned} Update \triangleq\ & (u,a,s_0,s_d,p_1^s,\ldots,p_m^s,p_1^g,\ldots,p_n^g).(k,s'_d) \\ & (\overline{u}\langle k\rangle.k(s'_0,s'_1,c,r',p').\overline{s_0}.\overline{a}\langle s'_0,s'_1,s'_d\rangle.\overline{s_d} \\ & .r'(p_1'^s,\ldots,p_n'^s) \\ & \quad .(x)(\overline{p_1^g}\langle x\rangle.x(p).\overline{p_1'^s}\langle p\rangle\ldots\overline{p_n^g}\langle x\rangle.x(p).\overline{p_n'^s}\langle p\rangle) \\ & .p'(p_1'^g,\ldots,p_m'^g) \\ & \quad .(x)(\overline{p_1'^g}\langle x\rangle.x(p).\overline{p_1^s}\langle p\rangle\ldots\overline{p_n'^g}\langle x\rangle.x(p).\overline{p_m^s}\langle p\rangle) \\ & .\overline{s'_1}.Update\lfloor u,a,s'_0,s'_d,p_1^s,\ldots,p_m^s,p_1^g,\ldots,p_n^g\rfloor) \end{aligned}$$

Process $Update$ sends via name $u$ a request for a clone of a component. A new component that is the clone of the requested component will be used in update as a replacement of the old subcomponent in a parent component implementing the update process (i.e. as its subcomponent). As a return value, process $Update$ receives a vector of names representing control interfaces for binding and referencing the new component's functional interfaces (see the process of cloning above). Name $a$ represents the parent component's internal control interface to attach the new component's stop and start interfaces ($s'_0$ and $s'_1$ names). Before the attaching, name $s_0$ is used to stop the old subcomponent and name $s_d$ to detach its stop and start interfaces. Finally, names $p_1^s,\ldots,p_m^s$, $p_1^g,\ldots,p_n^g$ represent a context of the old subcomponent, i.e. interfaces of neighbouring subcomponents, which have to be rebound to interfaces of the new component.

### 4.3.5 Primitive and Composite Components

Finally, we can describe complete behaviour of primitive and composite components. Let us assume that process abstraction $Comp_{impl}$ with parameters $s_0,s_1$, $r_1,\ldots,r_n$,

[6]The process is also known as "updating" or "nesting" of a component.

$p_1, \dots, p_m$ describes behaviour of the core of a primitive component (i.e. excluding behaviour of processing of its control actions), as it is defined by the component's developer. Further, process abstraction $Comp_{subcomps}$ with parameters $a$, $p'^s_1, \dots, p'^s_m$, $p'^g_1, \dots, p'^g_n$ describes behaviour of a system of subcomponents interconnected by means of their interfaces into a composite component (see Sec. 4.3.2). Names $s_0, s_1$, $r_1, \dots, r_n$, $p_1, \dots, p_m$ and names $a$, $p^s_1, \dots, p^s_m$, $p^g_1, \dots, p^g_n$ are defined in Sec. 4.3.1.

Processes $Comp_{prim}$ and $Comp_{comp}$ that describe behaviour of the mentioned primitive and composite components can be defined as follows:

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

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

where the pseudo-applications of $Ctrl_{Ifs}$ represent behaviour of control parts of the components related to their functional interfaces (see Sec. 4.3.2), the constant applications of $Ctrl_{clone}$ describe behaviour of control parts of the components related to their cloning (see Sec. 4.3.4), the pseudo-application of $Ctrl_{SS}$ represents behaviour of the composite component's control part processing its stop and start requests (see Sec. 4.3.3), and the pseudo-application of $Ctrl_{EI}$ describes communication between internal and external functional interfaces of the composite component (see Sec. 4.3.2).

## 4.4 Behavioural Modelling of Services

This section deals with linking individual services of SOA to their underlying implementations as CBSs. It provides an approach to formal description of these services as the CBSs by means of the component model from Sec. 4.

A system that applies SOA can be described at the following *three levels of abstraction*: as business processes that represent sequences of steps in accordance with some business rules leading to business aims; as services that implement the business processes with well-defined interfaces and interoperability for the benefit of business; and as components that implement the services as CBSs with well-defined structure and description of their behaviour. According to these three levels behaviour of a service can be described in two views:

1. The service is *an entity of* SOA *architecture* and is described by provided functionality and relations to its neighbouring services (the "services" level of abstraction). The neighbouring services can act as requesters of the service or providers of functionality required by the service. The service itself can also act as a parent service to the neighbouring services to ensure their assembly and coordination (i.e. as a "task-centric" service controlling service composition members, see [10]).
2. The service can be implemented as a CBS (the "components" level of abstraction). It is a component with external interfaces accessible by neighbouring components (neighbouring services at the "services" level of abstraction, i.e. independent requesters, providers, as well as potential service composition members). The component can be realised either as a primitive or as a composite component where the component's structure and its behaviour describe the service's internal implementation.

The first view requires description of the service's behaviour in the context of communication with its neighbouring services. The second shows the service as a component of CBS, which internal structure and behaviour can be specified in the common way, as in Sec. 4.

### 4.4.1 Service as a Part of SOA

The result of business-to-service transformation [32], which forms SOA services from business processes, is an UML class diagram. Individual *services* are modelled as UML classes with stereotype «`service`» and connected by means of UML relationships of "realisation" and "use" to UML classes with stereotype «`interface`». While the classes with stereotype «`service`» represent specific services, the classes with stereotype «`interface`» describe, by means of their methods, individual *interfaces* provided or required by the services (i.e. "services" provided or required by their "providers" or "consumers", respectively).

Let us assume a service `Service` that is described as an entity of SOA by its interfaces `I1` to `In` and relations to its neighbouring services (i.e. at the "services" level of abstraction and in the first view according to the introduction of Sec. 4.4). Behaviour of the service can be described as $\pi$-calculus process abstraction *Service*:

$$\begin{aligned}
Service \stackrel{def}{=}\ & (i_1, \dots, i_n).(b_1, \dots, b_m) \\
& (Svc_{init}\langle i_1, \dots, i_n, b_1, \dots, b_m\rangle \\
& .\prod_{j=1}^{n} Svc_j\langle i_j, b_1, \dots, b_m\rangle)
\end{aligned}$$

where names $i_1, \dots, i_n$ represent the service's interfaces `I1`, ..., `In`, respectively, the pseudo-application of $Svc_{init}$ initiates the service's behaviour, and the pseudo-application of $Svc_j$, for each $j \in \{1, \dots, n\}$, describes behaviour of processing of requests via the service's interface represented by name $i_j$ including possible communication via shared names $b_1, \dots, b_m$.

**Communication of Services and Service Broker.** Communication of services in SOA is realised by means of various styles of data passing. In a case of existing *service choreography or orchestration* in SOA, roles of participating services are predefined and the architecture is static. Then, the choreography or orchestration is described by means of a composition of $\pi$-calculus processes representing individual services, which communicate directly via names that represent the services' interfaces and that are shared among the processes.

However, a serious SOA will likely discover its services throughout an enterprise and beyond [10]. To support the dynamic service discovery and invocation, SOA provides service brokers (e.g. UDDI registries), which allow to publish, find, and bind services at runtime.

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

$$
\begin{aligned}
Broker &\stackrel{def}{=} (pub, find).(q) \\
&\quad (Publish\lfloor q, pub\rfloor \mid Find\lfloor q, find, pub\rfloor) \\
Publish &\triangleq (t, pub).pub(i, d).(t') \\
&\quad (\bar{t}\langle t', i, d\rangle \mid Publish\lfloor t', pub\rfloor) \\
Find &\triangleq (h, find, pub).h(h', i, d) \\
&\quad .(Find\lfloor h', find, pub\rfloor \mid (\overline{find}\langle i\rangle.\overline{pub}\langle i, d\rangle \;+\; d))
\end{aligned}
$$

where names representing the providers' interfaces (denoted by $i$ internally) can be stored via name $pub$ and retrieved back via name $find$, which are subsequently handled by constant applications of $Publish$ and $Find$, respectively. By the composition of the constant applications of $Publish$ and $Find$ with shared name $q$, process constant $Broker$ implements basic operations on a simple queue (i.e. a First-In-First-Out (FIFO) data structure).

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

The constant application of $Find$ dequeues a front item of the queue as a triple of names $(h', i, d)$ via name $h$, that represents *the current head of the queue*. Then, it proceeds as a composition of a constant application of $Find\lfloor h', find, pub\rfloor$, which handles future requests, and a sum of capabilities of process $\overline{find}\langle i\rangle.\overline{pub}\langle i, d\rangle$, which provides name $i$ as an interface for potential service requesters and enqueues it back to the queue via name $pub$, 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.

### 4.4.2 Service as a Component-Based System

A service's underlying implementation, its behaviour, and internal structure, can be described as a CBS. The service can be implemented as *a component with external provided and required interfaces*, which correspond to the services' interfaces provided to its possible consumers and required from other services to consume their functionality, respectively. This approach is related to the "components" level of abstraction and the second view from the introduction of Sec. 4.4.

To describe a service `Service` with interfaces `I1` to `In` as a CBS and by means of the component model from this article (see Sec. 4), we need to transform $\pi$-calculus process abstraction $Service$ from Sec. 4.4.1 describing behaviour of the service into a formal description of behaviour of a component representing the CBS (see Sec. 4.3). We focus on pseudo-application $Svc_j\langle i_j, b_1, \ldots, b_m\rangle$, which describes specific processing of the service's interface $i_j$ (for $j \in \{1, \ldots, n\}$) and communication with other parts of the service via shared names $b_1, \ldots, b_m$. Process abstraction $Svc_j$ can be defined as follows:

$$
\begin{aligned}
Svc_j &\stackrel{def}{=} (i_j, b_1, \ldots, b_m). \\
&\quad Svc'_j\langle i_j, b_{x_1}, \ldots, b_{x_k}, b_{y_1}, \ldots, b_{y_{(m-k)}}\rangle
\end{aligned}
$$

where $k \in \{1, \ldots, m\}$ and $x_1, \ldots, x_k, y_1, \ldots, y_{(m-k)} \in \{1, \ldots, m\}$, and sets $\{b_{x_1}, \ldots, b_{x_k}\} \cap \{b_{y_1}, \ldots, b_{y_{(m-k)}}\} = \emptyset$ and $\{b_{x_1}, \ldots, b_{x_k}\} \cup \{b_{y_1}, \ldots, b_{y_{(m-k)}}\} = \{b_1, \ldots, b_m\}$ (see the pseudo-application of $Svc_j$ in Sec. 4.4.1).

Name $i_j$ represents the interface `Ij` provided by the service, names $b_{x_1}, \ldots, b_{x_k}$ are all of the shared names that are used as channels of input prefixes in $Svc'_j$ and names $b_{y_1}, \ldots, b_{y_{(m-k)}}$ are all of the shared names that are used as channels of output prefixes in $Svc'_j$ (for input and output prefixes, see Sec. 4.1). Thereafter, process abstraction $Svc'_j$ can be understand as a description of core behaviour of a component with functional provided interfaces represented by names $i_j, b_{x_1}, \ldots, b_{x_k}$ and functional required interfaces represented by names $b_{y_1}, \ldots, b_{y_{(m-k)}}$ in the external view (see Sec. 4.3).

The mentioned component implements a part of the service that is related to its interface `Ij` as a CBS. To extract the desired core behaviour from the component's complete behaviour, process abstraction $Svc'_j$ can be defined as:

$$
\begin{aligned}
Svc'_j &\stackrel{def}{=} (i_j, b_{x_1}, \ldots, b_{x_k}, b_{y_1}, \ldots, b_{y_{(m-k)}}). \\
&\quad (s_0, s_1, c, p^s_1, \ldots, p^s_{(m-k)}, p^g_1, \ldots, p^g_{(k+1)}) \\
&\quad (\textstyle\prod_{u=1}^{k}(d, t)(\overline{p^g_{(u+1)}}\langle t\rangle.t(p).Wire\lfloor b_{x_u}, p, d\rfloor) \\
&\quad \mid \textstyle\prod_{v=1}^{m-k}\overline{p^s_v}\langle b_{y_v}\rangle \mid (d, t)(\overline{p^g_1}\langle t\rangle.t(p).Wire\lfloor i_j, p, d\rfloor) \\
&\quad \mid Comp_j\langle s_0, s_1, c, p^s_1, \ldots, p^s_{(m-k)}, p^g_1, \ldots, p^g_{(k+1)}\rangle)
\end{aligned}
$$

where process constant $Wire$ has been defined in Sec. 4.3.2 and process abstraction $Comp_j$ describes the component's complete behaviour and is fully compatible with behavioural description of primitive and composite components from Sec. 4.3.5.

## 5. Main Contributions

The proposed component model and the behavioural modelling have been successfully validated in a case study of a SOA of an environment for functional testing of complex safety-critical systems, which has been published in [30] and in the author's dissertation [31].

Current approaches related to our work can be divided into two groups as follows:

1. *formal approaches to modelling of* SOA*s*, mostly based on the formalisation of business process models (e.g. transformations of BPEL to Petri nets or to $\pi$-calculus processes);
2. *formal approaches to modelling of* CBS*s*, such as component models and ADLs mentioned in Sec.2.2, which are usually focused only on CBSs without consideration of SOA at the higher level of abstraction (e.g. Wright [1], Tracta [12], behaviour protocols of SOFA [37], formal descriptions of Fractive [5], and, partially, SOFA 2.0 [8]).

Our approach intends to bridge the gap and to provide a formal description of SOAs from the choreography of their services to the behaviour of individual components of underlying CBSs, as it has been demonstrated in the case study. Similar efforts can be found in SOFA 2.0 and the Reo coordination language [9].

In the SOFA 2.0, SOA becomes a specific case of a CBS where all components (services) are interconnected solely via their utility interfaces. The interfaces can be referred and freely passed among the components and used to establish new connections, independently of levels of component hierarchy. The Reo coordination language [2, 9] is based on the $\pi$-calculus and able to describe coordination of both services in SOA and components in CBSs.

In comparison with SOFA 2.0 or the Reo coordination language, our approach describes services and components separately and with respect to their differences (i.e. services are not components and vice versa), but it allows to go smoothly from a service level to a component level and to describe behaviour of a whole system, services and components, as a single $\pi$-calculus process. Moreover, we use the polyadic $\pi$-calculus without any special extensions, which allows us to utilise existing tools for model checking of $\pi$-calculus processes and verification of their properties (e.g. *The Mobility Workbench* [36], a model checker and an open bisimulation checker of mobile concurrent systems described in the $\pi$-calculus).

Generally, in comparison to the current approaches, our approach has the following *important merits*:

- The proposed component model has been *designed for mobile architectures*. It supports fully dynamic architectures with component mobility.
- The model permits *combination of control and functional interfaces* in behaviour of primitive components. Dynamic reconfiguration and component mobility can be initiated by functional requirements and performed via the control interfaces.
- Behaviour of services and components is described in the $\pi$-calculus, which has a *native support for reconfiguration and mobility*. It is a suitable formal basis for behavioural description of systems with mobile architectures.
- We use the *polyadic $\pi$-calculus* without any special extensions, which allows us to utilise existing tools for model checking of $\pi$-calculus processes and formal verification of their properties.
- The proposed *behavioural modelling of* SOA*s* allows a developer to go from a high level service design to a detailed design of underlying CBSs, with respect to differences between services and components. Behaviour of a whole system (individual services, their choreography and implementation as the CBSs) can be described as a single $\pi$-calculus process.

However, the proposed approach can suffer from the following *possible drawbacks*:

- The behavioural description of services and components in $\pi$-calculus *uses infinite recursions*. These are implemented by unguarded or weakly guarded applications and which can cause decidability issues.
- The representation of system models uses the *specific and informal* UML-*like notation*.
- The formal description of behaviour *requires an advanced knowledge of the $\pi$-calculus* and may be a difficult task for unskilled developers.
- The proposed approach describes how to model a specific configuration and behaviour of a CBS or a SOA as a $\pi$-calculus process. However, after several dynamic reconfigurations and a corresponding sequence of reductions of the $\pi$-calculus process, it may be difficult to determine a final configuration from the resulting $\pi$-calculus process, especially without knowledge of the exact sequence of reductions (e.g. it may be difficult to determine a deadlock configuration, which has been detected by means of a verification tool in a specific $\pi$-calculus process).

## 6. Conclusions

In this article, the approach for modelling of CBSs has been proposed. It meets the objectives set out in Sec. 3.2. We have presented the component model, which allows to describe CBSs with mobile architectures (i.e. dynamic architectures allowing component mobility). The component model's metamodel has been introduced to describe basic entities of the component model and their relations and features. We have also proposed the formal description of behaviour of the component model's entities and services of SOAs as $\pi$-calculus processes. It allows us to pass smoothly from service level to component level and to describe behaviour of a whole system, services and components, as a single $\pi$-calculus process.

An application of our approach has been illustrated in the case study of the environment for functional testing of complex safety-critical systems, which has been published in [30]. In the case study, the environment has been described as a SOA and an underlying CBS modelled by means of the component model's metamodel. We have formally described behaviour of the whole environment by means of the $\pi$-calculus on the levels of the SOA and the CBS. Finally, the formally described services and components have been simulated, checked for deadlocks, strong and weak open bisimulation equivalence, and verification of their properties has been outlined.

In comparison with the related approaches, the proposed approach has advantages in support of mobile architectures, in full integration of dynamic reconfiguration into behaviour of components where functional requirements can initiate control actions, in support of behavioural description of SOAs and transition to CBSs, and in utilisation of the standard polyadic $\pi$-calculus supported by existing tools for model checking and formal verification.

The future work will be focused on improving system models' notation, modelling tools, and behavioural description, to simplify integration of the component model and utilisation of its formal basis into initial phases of software development processes. We also intend to support final phases of the development processes by integration of the component model into existing component-based technologies and by an implementation framework.

**Acknowledgements**. This research was partially supported by the BUT FIT grant FIT-10-S-2 and the Research Plan No. MSM 0021630528 – Security-Oriented Research in Information Technology.

## References

[1] R. Allen and D. Garlan. The Wright architectural specification language. Technical Report CMU-CS-96-TB, Carnegie Mellon University, School of Computer Science, Pittsburgh, 1996.

[2] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[3] P. Avgeriou, N. Guelfi, and N. Medvidovic. Software architecture description and UML. In *UML Satellite Activities*, volume 3297 of *Lecture Notes in Computer Science*, pages 23–32. Springer, 2004.

[4] D. Balasubramaniam, R. Morrison, F. Oquendo, I. Robertson, and B. Warboys. Second release of ArchWare ADL. Technical Report D1.7b (and D1.1b), ArchWare Project IST-2001-32360, 2005.

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

[6] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, June 2002.

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

[8] T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006*, pages 40–48, Seattle, USA, 2006. IEEE Computer Society.

[9] N. K. Diakov and F. Arbab. Compositional construction of Web Services using Reo. In *Proc. of International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI 2004)*, pp. 49–58. INSTICC Press, 2004.

[10] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[11] D. Garlan, R. T. Monroe, and D. Wile. ACME: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pp. 47–68. Cambridge University Press, NY, 2000.

[12] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology and Medicine University of London, Department of Computing, 1999.

[13] D. Hatebur, M. Heisel, and J. Souquières. A method for component-based software and system development. In *Proceedings of the 32nd EUROMICRO Conference on Sotware Engineering and Advanced Applications*, pages 72–80. IEEE Computer Society, 2006.

[14] P. Hnětynka and F. Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of CBSE 2006*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.

[15] Recommended practice for architectural description of software intensive systems. Technical Report IEEE P1471–2000, The Architecture Working Group of the Software Engineering Committee, Standards Department, IEEE, Piscataway, New Jersey, USA, 2000.

[16] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[17] K.-K. Lau and Z. Wang. A taxonomy of software component models. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 88–95. IEEE Computer Society, 2005.

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

[19] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[20] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, 2002.

[21] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

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

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

[24] Meta object facility (MOF) core specification, ver. 2.0. Document formal/06-01-01, The Object Management Group, 2006.

[25] UML infrastructure, version 2.1.2. Document formal/2007-11-04, The Object Management Group, 2007.

[26] UML superstructure, version 2.1.2. Document formal/2007-11-02, The Object Management Group, 2007.

[27] F. Oquendo. $\pi$-ADL: an architecture description language based on the higher-order typed $\pi$-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29:1–14, 2004.

[28] SCA service component architecture: Assembly model specification. Technical Report SCA version 1.00, Open SOA Collaboration, 2007.

[29] 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, 1998. IEEE Computer Society.

[30] M. Rychlý. A case study on behavioural modelling of service-oriented architectures. In *Software Engineering Techniques In Progress*, pp. 79–92. AGH University Press, 2009.

[31] M. Rychlý. *Formal-based Component Model with Support of Mobile Architecture*. PhD thesis, Department of Information Systems, Faculty of Information Technology, Brno University of Technology, 2010.

[32] M. Rychlý and P. Weiss. 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*, pages 140–146. Institute for Systems and Technologies of Information, Control and Communication, 2008.

[33] D. Sangiorgi and D. Walker. *The $\pi$-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New Ed edition, 2003.

[34] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 2nd edition, 2002.

[35] S. Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, 1993.

[36] B. Victor. *The Mobility Workbench User's Guide*, polyadic version 3.122 edition, 1995.

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

## Selected Papers by the Author

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, 2006.

M. Rychlý and J. Zendulka. Distributed information system as a system of asynchronous concurrent processes In *MEMICS 2006 Second Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 206–213. Faculty of Information Technology BUT, 2006.

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, 2007.

M. Rychlý and P. Weiss. 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*, pages 140–146. Institute for Systems and Technologies of Information, Control and Communication, 2008.

M. Rychlý. Behavioural modeling of services: from service-oriented architecture to component-based system In *Software Engineering Techniques in Progress*, pages 13–27. Wroclaw University of Technology, 2008.

M. Rychlý. A component model with support of mobile architectures and formal description *e-Informatica Software Engineering Journal*, 3(1):9–25, 2009.

M. Rychlý. A case study on behavioural modelling of service-oriented architectures *e-Informatica Software Engineering Journal*, 4(1):71–87, 2010.