BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PHD THESIS
TEZE DISERTAČNÍ PRÁCE

FORMAL-BASED COMPONENT MODEL WITH SUPPORT OF MOBILE ARCHITECTURE

FORMÁLNÍ KOMPONENTOVÝ MODEL PRO MOBILNÍ ARCHITEKTURY

MGR. MAREK RYCHLÝ

SUPERVISOR: DOC. ING. JAROSLAV ZENDULKA, CSC.

Contents

Abstrakt (in Czech) **2**

Abstract **2**

1 State of the Art **3**
1.1 Software Architecture and Component-Based Development . . . 3
1.2 Component Models and Architecture Description Languages . . . 3

2 Motivation **5**
2.1 Statement of the Problem . . . 5
2.2 Objectives of the Thesis . . . 5

3 Methods of Realisation **5**
3.1 Formal Base . . . 6
3.2 Component Model: Logical View . . . 6
3.2.1 Metamodel . . . 7
3.2.2 System Model . . . 11
3.3 Component Model: Process View . . . 13
3.3.1 Notation . . . 13
3.3.2 Interface's References and Binding . . . 13
3.3.3 Control of a Component's Life-cycle . . . 14
3.3.4 Cloning of Components and Updating of Subcomponents . . . 14
3.3.5 Primitive and Composite Components . . . 15
3.4 Behavioural Modelling of Services . . . 16
3.4.1 Service as a Part of Service-Oriented Architecture . . . 16
3.4.2 Service as a Component-Based System . . . 17
3.5 Development Process . . . 17
3.5.1 Application of the Behavioural Modelling of Services . . . 17
3.5.2 Application of the Component Model . . . 18
3.5.3 Integration of a Formal Description . . . 19

4 Main Contributions **19**

5 Conclusions **20**

References **20**

Abstrakt (in Czech)

Globalizace a rozvoj informační společnosti vyžadují rozsáhlá a spolehlivá IT řešení. Požadavky na současné informační systémy často výrazně ovlivňují jejich architekturu. Systémy pak nemohou být realizovány jako celky, ale jako sítě oddělených a spolupracujících komponent, které komunikují asynchronně pomocí zpráv předem definovaných formátů. Volné vazby těchto komponent umožňují za běhu dynamicky navazovat a rušit jejich vzájemné propojení, podle aktuálních potřeb a omezení (např. podle kvality a ceny komponentami poskytovaných a požadovaných služeb), kopírovat komponenty a přesouvat je do jiných kontextů (tzv. mobilita komponent), nebo dynamicky vytvářet, odstraňovat a aktualizovat komponenty.

Zmíněné dynamické aspekty softwarových architektur a mobilita komponent přinášejí nové problémy. Komponentové systémy se stávají příliš složitými a je potřeba formální popis evoluce jejich architektur, zejména v případě kritických aplikací. Návrh takových systémů nelze vést standardními metodami.

Disertační práce se zabývá modelováním komponentových systémů a formálním popisem jejich chování. Řešení je založeno na vlastním komponentovém modelu, který je popsán meta-modelem, z logického pohledu, a popisem v π-kalkulu, z procesního pohledu. Je ukázáno, že komponentový model pokrývá dynamické aspekty softwarových architektur včetně mobility jejich komponent. Dále je popsán způsob modelování chování v architekturách orientovaných na služby a přechod ke komponentovým systémům. Chování konkrétní architektury orientované na služby lze pak vyjádřit jako jediný proces v π-kalkulu. V závěru práce je navržené řešení ověřeno na případové studii prostředí pro testování kritických aplikací. Přínosem disertační práce je zejména zmíněná podpora dynamických architektur a integrace s architekturami orientovanými na služby.

Abstract

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 and cost 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 brings new problems in the domain of software engineering. The component-based systems are getting involved, and a formal specification of evolution of their architectures is necessary, particularly in critical applications. Design of these systems can not be done by means of conventional software design methods.

In the thesis, we propose an approach to modelling of component-based systems and formal description of their behaviour. The approach is based on a novel component model defined by a metamodel in a logical view and by description in the π-calculus in a process view. We show that the component model addresses the dynamic aspects of software architectures including the component mobility. Furthermore, we propose a method of behavioural modelling of service-oriented architectures to pass smoothly from service level to component level and to describe behaviour of a whole system, services and components, as a single π-calculus process. Finally, we illustrate an application of our approach on a case study of an environment for functional testing of complex safety-critical systems. The support of dynamic architecture and the integration with service-oriented architecture compromise the main advantages of our approach.

1 State of the Art

This section deals with software architecture and component-based development. It provides an overview of the state of the art of component models and architecture description languages. We focus on dynamic and mobile architectures.

1.1 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“ [21] can be described using several concurrent views [22, 21] including particularly logical (structural) view and process (behavioural) view. The *logical view* describes logical structure of the system, while the *process view* describes concurrency and synchronisation aspects of the system, e.g. behaviour of components as processes, communication constraints, evolution of the system in time, etc.

We can distinguish three types of software architectures according to their evolution in dependence on changes of their environment [38]: 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 the system's design-time. Finally, the *mobile architecture* is a dynamic architecture of a system where the system's components can change their context in the system's logical structure during its execution (also called „component mobility“) according to rules from the system's design-time and functional requirements.

The **component-based development** (CBD, see [43, 13]) is a software development methodology, which is strongly oriented to composability and re-usability in a software system's architecture. In the CBD, from a structural point of view, a *component-based system* is composed of *components*, which are self-contained entities accessible through well-defined *interfaces*. A connection of compatible interfaces of cooperating components is realised via their *bindings* (connectors). Actual organisation of interconnected components is called *configuration*.

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, in other words a *reconfiguration*. Description of the reconfiguration [38] 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 (they are „black-boxes“). The *composite components* are decomposable into systems of subcomponents at the lower level of architecture description (they are „grey-boxes“). This composition forms *a component hierarchy*.

1.2 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 [23]. 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 [24] including the models, which are mentioned in this section. 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 behavioural descriptions in CSP support only systems with static architectures).

Component model Darwin [26, 17] 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 [26] allows to describe behaviour of components in the calculus of mobile processes (the π-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 [17] 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* [40], 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, [29]), which extends features of OMG IDL [32] to allow specification of software components. Behaviour of a component (its interface, frame, and architecture) is formally described by means of *behaviour protocols* [46]. 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 [11], 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 [9, 10] 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 [7]. 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 [44]) 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 Section 1.1). Several ADLs have been proposed [28]: 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 [16] 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 [34, 37]) provides three possible strategies for modelling of software architectures [27]: 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 component-based systems [5] by means of *structured classes*, i.e. the classes that allow nesting of other classes. Its specification [37] 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 [4].

ArchWare ADL (see [6]) 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: π-ADL [38] that contains the core structure and behaviour constructs with the higher-order typed π-calculus as a formal basis; $\sigma\pi$-ADL [6] 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 μ-calculus for description of behavioural and structural properties of communicating and mobile architectural elements. The behaviour constructs from the base language copy the π-calculus constructs.

2 Motivation

This section aims at clarifying the motivation of the thesis. A problem is stated in terms of the state of the art and objectives of the thesis are established.

2.1 Statement of the Problem

The current component models and architecture description languages have many shortcomings in support of mobile architectures, incorporation of component-based design into service-oriented architecture and into software development processes in general. Their formal bases or models usually do not consider component mobility (e.g. PNETs in Fractal [7], behaviour protocols in SOFA [46], and reconfiguration patterns in SOFA 2.0 [20]), 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 [7]), do not support description of service-oriented architectures where individual services can be implemented as underlying component-based systems (e.g. in Fractal component model [10] or in the ArchWare project [3]), etc.

Moreover, the component models do not describe related methods of component-based development and their incorporation into well-established software development processes of standard software systems [18]. Modelling of the component-based systems during the development processes brings many issues (e.g. as a consequence of different conceptions of components in the component models [23], in component diagrams of UML [34, 37] or software component architectures [39]).

2.2 Objectives of the Thesis

The statement of the problems of current component models and corresponding architecture description languages provides us with adequate motivation. A *general objective of the thesis* is to design a component model for mobile architectures. The model has to provide suitable formal basis and should be applicable to modelling of component-based systems as well as service-oriented architectures.

Specific objectives of the thesis 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 component-based development; integration of functional operations and relevant behaviour of components with control operations enabling dynamic reconfiguration; a method of application of the component model in service-oriented architectures with mapping rules between services and component-based systems 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.

3 Methods of Realisation

In order to reach the stated objectives, we propose a high-level component model addressing the current issues of component-based development. 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 component-based systems (CBSs) is based on the π-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 and features. The second view, is focused on behaviour of the component model's entities, especially on component mobility. Finally, we describe behaviour of services in service-oriented architectures (SOAs) and their underlying implementations as component-based systems.

3.1 Formal Base

To describe behaviour of components in CBSs and services in SOAs in formal way, we use the π-calculus, known also as a *calculus of mobile processes* [30]. 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, 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 π-calculus, a theory of mobile processes, according to [42]:

- $\overline{x}\langle y\rangle.P$ 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 $P\{y/x\}$.

3.2 Component Model: Logical View

The component model for mobile architectures is described as a metamodel in the context of a four-layer modelling architecture [33]. The metamodel is implemented in OMG's *Meta Object Facility* (MOF, [35]), 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. This layer contains particular instances of component-based systems, 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 component-based systems such as, for example, UML component models or communication diagrams.
- 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.

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

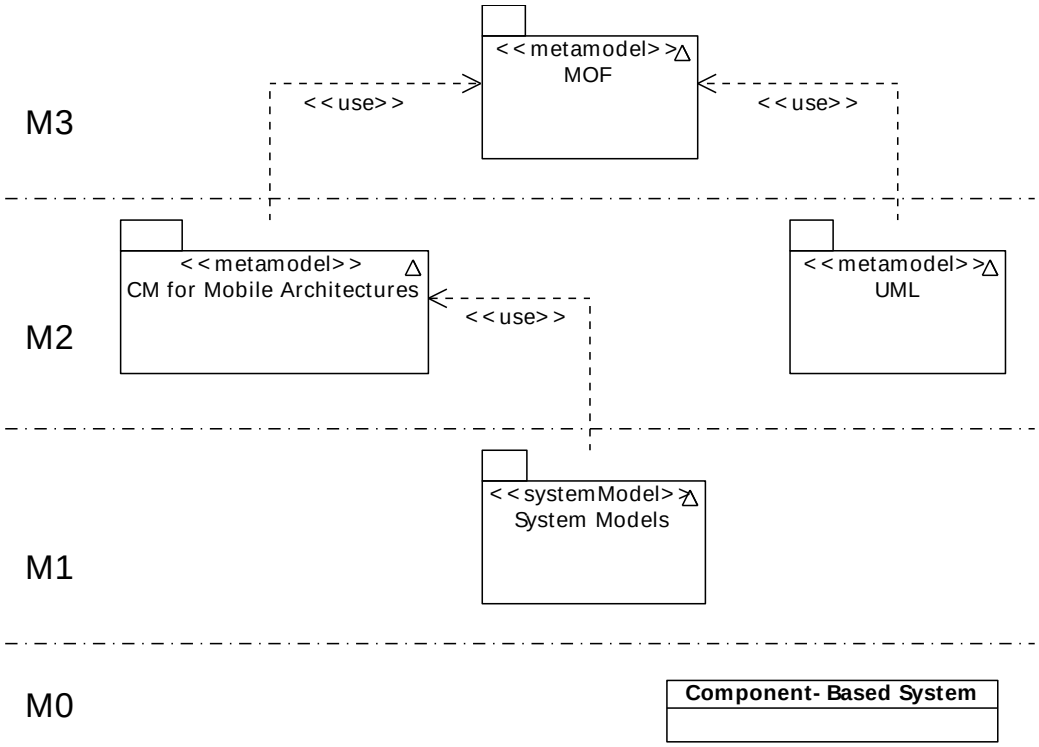

Figure 1: The four-layer modelling architecture of the component model and UML as metamodels in layer M2 and MOF as a meta-metamodel in layer M3 (UML 2 notation).

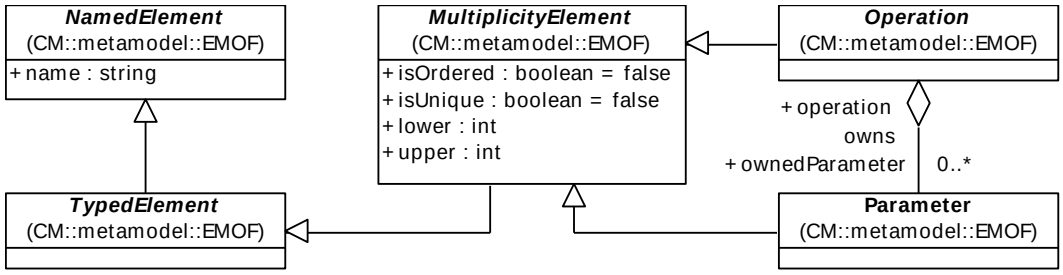

Figure 2: A simplified part of the EMOF metamodel [35] with classes that will be extended by the component model.

The four-layer modelling architecture is shown in Figure 1. Between models of layer M1, layer M2, and layer M3, there is a relationship denoted by a dependency with UML 2 stereotype «`use`», i.e. the models in lower layers use classes from metamodels in upper layers to create their objects. In the context of component-based development, a specific component-based system (layer M0) contains instances of elements from its model (stereotyped as «`systemModel`» in 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), both with stereotypes «`metamodel`».

3.2.1 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, [35]). MOF is in layer M3 in the four-layer modelling architecture (see Section 3.2). 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 [36]). For purposes of this thesis, 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:

1. 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“,
2. 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`,

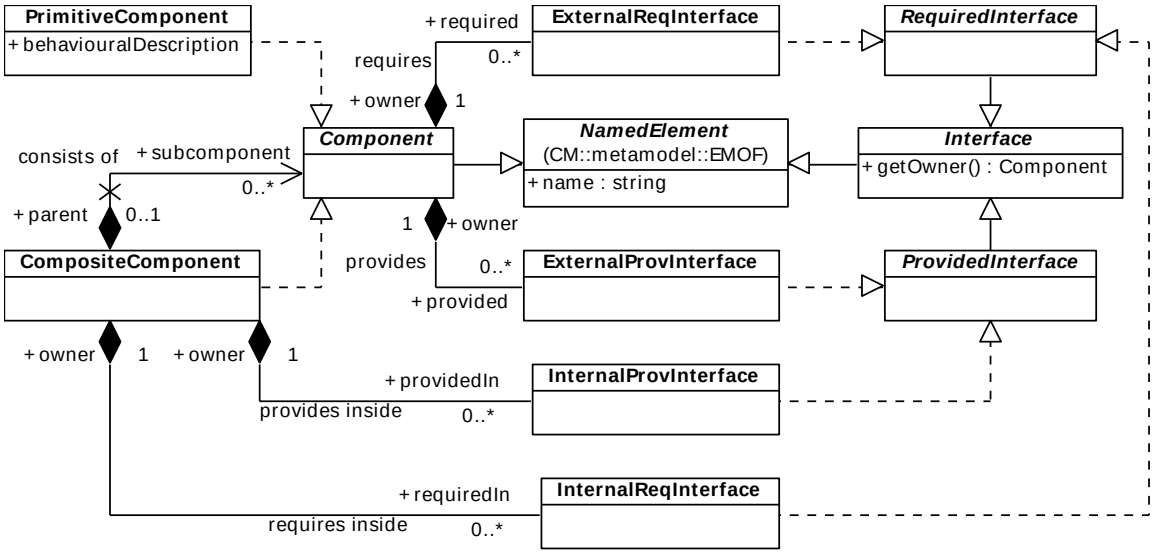

Figure 3: Abstract component, realisations, and interfaces, extending `EMOF::NamedElement` in the metamodel of the component model.

`EMOF::TypedElement`, and `EMOF::Operation`, which are outlined in Figure 2. A complete and detailed definition of the EMOF classes can be found in [35].

Components and Interfaces: Figure 3 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*, which is an active communicating entity of a component-based software system, can be described form two sides: as an abstract component without considering its internal structure („black-box“ view) and as a component realisation in the form of a primitive component or a composite component („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`) is decomposable on a system of subcomponents at the lower level of architecture description (it is a „grey-box“). 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 (in accordance with `owner` roles of components in the relations with their interfaces).

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 Figure 4. 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[2]. The binding interconnects required interfaces (class `RequiredInterface`) via relations „bound from“ to provided interfaces (class `ProvidedInterfaces`) via relations „bound to“.

[2]The diagram in Figure 4 does not restrict relations of `BindSiblings` to the interfaces of the same composite component realisations; this will be defined later by means of additional constraints.

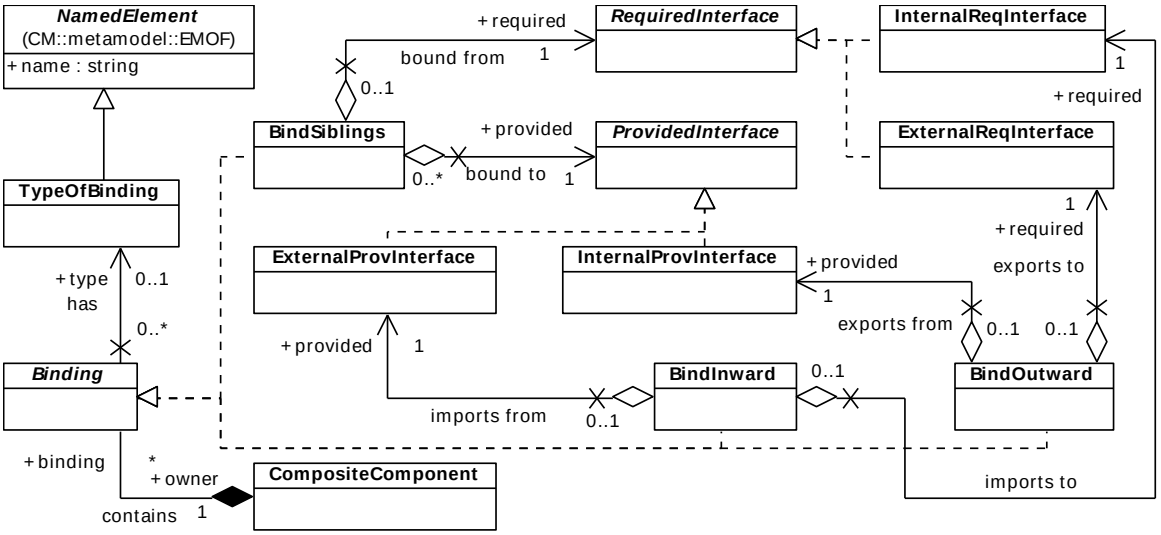

Figure 4: Binding and its different realisations between interfaces of a composite component realisation in the metamodel of the component model. Classes `CompositeComponent` and `...Interface` are identical to the classes in Figure 3.

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 class `ExternalReqInterfaces`).

The bindings (i.e. instances of the realisations of class `Binding`) are owned by the composite component realisations. Each binding can have a type (class `TypeOfBinding`), a specialisation of `EMOF::TypedElement`, 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 Figure 5.

According to a scope of visibility of the interfaces in a composite component realisation, we can distinguish public interfaces, private interfaces, 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 binding `BindInward` and `BindOutward` owned by the component). 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[3]. 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, but if the component is a composite component realisation, they are not reachable by its subcomponents. They can be interconnected only by means of binding `BindSiblings`.

According to functionality, we can distinguish the following types of interfaces[4] (see Figure 5):

- 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.

[3]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). It means that the subcomponents have to be primitive components.

[4]Type `Operation` denotes *functional interfaces*, while the others denote *control interfaces*.

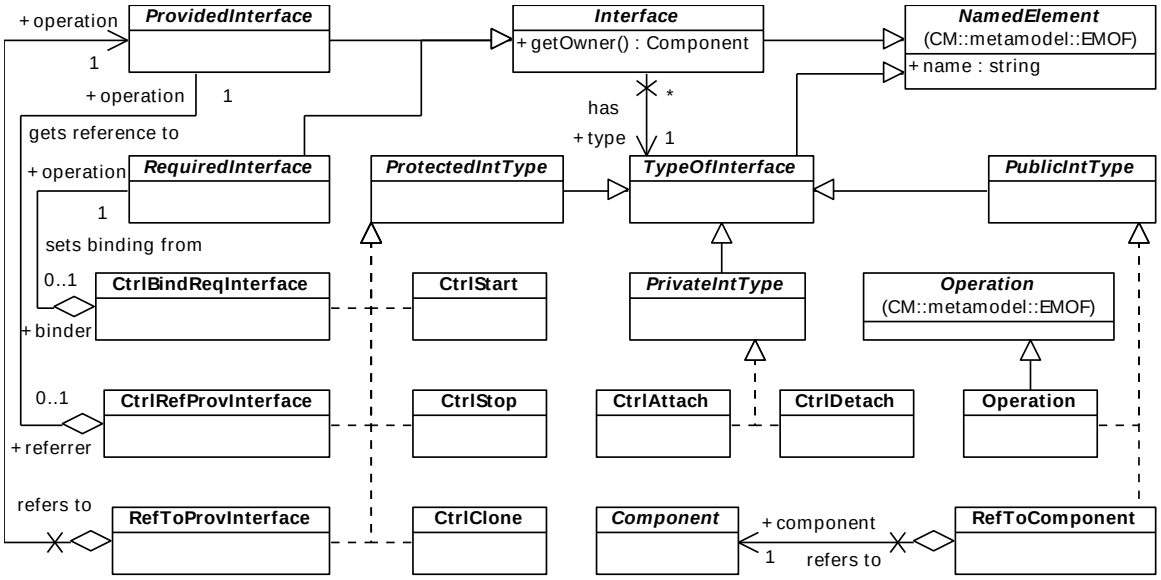

Figure 5: Types of interfaces with class `Operation` extending `EMOF::Operation` in the metamodel of the component model. Classes `Interface`, `ProvidedInterface`, `RequiredInterface`, and `Component` are identical to the classes in Figure 3.

- Protected interface `CtrlRefProvInterface` provides references to given provided interface `ProvidedInterface` of type `Operation`[5], while protected interface `CtrlBindReqInterface` allows to establish a new binding of specific required interface `RequiredInterface` of type `Operation`[5] 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 of a new component as a subcomponent of the composite component realisation („nesting“ of the component) and detach of 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`[5], while public interface `RefToComponent` allows to pass references of a whole component `Component`, which is required to support component mobility.

Additional Constraints: We need to define additional constraints to ensure type compatibility of interfaces in bindings, i.e. instances of realisations of class `Binding` in Figure 4. Types of the interfaces are given by relation to specific instances of realisations of class `TypeOfInterface` and according to the hierarchy of the types of interfaces in Figure 5. The following formulae use a first-order logic with extra predicate symbols „o : T“ and „o is T“ for restriction of o to type `T`, predicate symbol „$i \in$ L“ for restriction of l to list `L`, predicate symbol „$x = y$“ to check equality of x and y, and function symbol „i. getOwner()“ to get an owner of interface `i` (see method `getOwner()` of `Interface` in Section 3.2.1).

1. Bindings `BindInward` and `BindOutward` in a composite component realisation can interconnect only interfaces of the same composite component realisation.

$$
\begin{aligned}
&(\forall c : \text{CompositeComponent})\ (\\
&((\forall b : \text{BindInward} \in c.\,\text{binding})(b.\,\text{provided}.\,\text{getOwner}() = c \ \wedge\ b.\,\text{required}.\,\text{getOwner}() = c)) \wedge \\
&((\forall b : \text{BindOutward} \in c.\,\text{binding})(b.\,\text{provided}.\,\text{getOwner}() = c \ \wedge\ b.\,\text{required}.\,\text{getOwner}() = c))\ )
\end{aligned}
$$

2. Binding `BindSiblings` in a composite component realisation can interconnect only internal interfaces of the same composite component realisation or external interfaces of its subcomponents.

$$
\begin{aligned}
&(\forall c : \text{CompositeComponent})\ (\forall b : \text{BindSiblings} \in c.\,\text{binding})\ (\\
&\quad (\forall i : \text{InternalProvInt} \in b.\,\text{provided})\ (i.\,\text{getOwner}() = c) \\
&\quad \wedge\ (\forall i : \text{InternalReqInt} \in b.\,\text{required})\ (i.\,\text{getOwner}() = c) \\
&\quad \wedge\ (\forall i : \text{ExternalProvInt} \in b.\,\text{provided})\ (i.\,\text{getOwner}() \in c.\,\text{subcomponent}) \\
&\quad \wedge\ (\forall i : \text{ExternalReqInt} \in b.\,\text{required})\ (i.\,\text{getOwner}() \in c.\,\text{subcomponent})\ )
\end{aligned}
$$

3. Bindings `Binding` in a composite component realisation can interconnect only provided interfaces with required interfaces of compatible types.

[5]The restriction to the interface of type `Operation` will be defined explicitly by additional constraints.

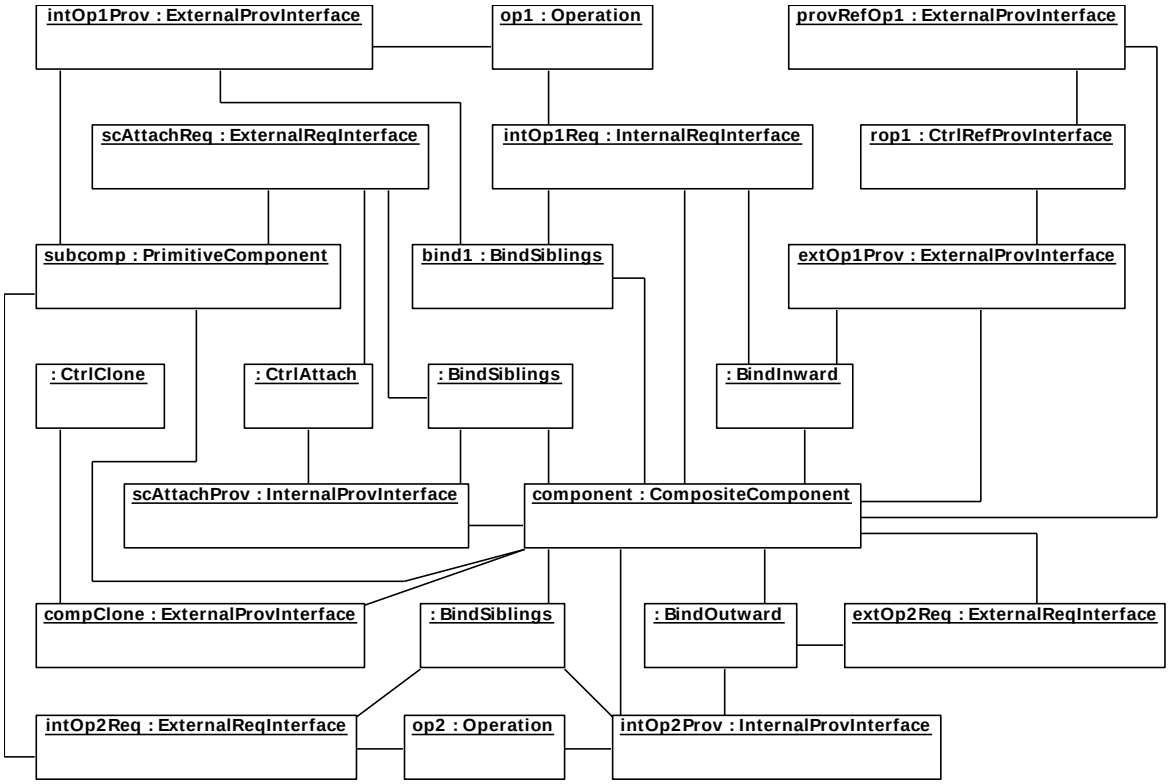

Figure 6: The example of description of a system model as an object diagram with instances of classes from the component model's metamodel.

$$(\forall c : \text{CompositeComponent})\ (\forall b : \text{Binding} \in c.\,\text{binding})(b.\,\text{provided.}\,\text{type} = b.\,\text{required.}\,\text{type})$$

4. Bindings `BindInward` and `BindOutward` can interconnect only public interfaces, i.e. instances of class `PublicIntType`.

$$\begin{array}{l}(\forall c : \text{CompositeComponent})\ (\\ \quad((\forall b : \text{BindInward} \in c.\,\text{binding})\\ \qquad(b.\,\text{provided.}\,\text{type is PublicIntType} \wedge b.\,\text{required.}\,\text{type is PublicIntType})\ ) \wedge\\ \quad((\forall b : \text{BindOutward} \in c.\,\text{binding})\\ \qquad(b.\,\text{provided.}\,\text{type is PublicIntType} \wedge b.\,\text{required.}\,\text{type is PublicIntType})\ )\ )\end{array}$$

5. Bindings `BindSiblings` that are inside a composite component realisation can be connected to private interfaces, only if the interfaces are internal interfaces of the composite component realisation.

$$\begin{array}{l}(\forall c : \text{CompositeComponent})\ (\forall b : \text{BindSiblings} \in c.\,\text{binding})\\ \quad(b.\,\text{provided.}\,\text{type is PrivateIntType} \Rightarrow b.\,\text{provided} \in c.\,\text{providedIn})\end{array}$$

6. Instances of classes `CtrlBindReqInterface`, `CtrlRefProvInterface`, and `RefToProvInterface`, and their relations to interfaces via „sets binding from", „gets reference to" and „refers to", respectively, have to be connected with the interfaces of type `Operation` only.

$$\begin{array}{l}(\forall t : \text{CtrlBindReqInterface})\ (t.\,\text{operation.}\,\text{type is Operation})\\ \wedge\ (\forall t : \text{CtrlRefProvInterface})\ (t.\,\text{operation.}\,\text{type is Operation})\\ \wedge\ (\forall t : \text{RefToProvInterface})\ (t.\,\text{operation.}\,\text{type is Operation})\end{array}$$

3.2.2 System Model

The component model's metamodel is a model of layer M2 where it exists alongside UML (see Figure 1 in Section 3.2). Both the component model and UML are based on the same meta-metamodel MOF in layer M3, although they are distinct in purpose and also in practice. In this section, we utilise the notation of UML 2 component diagrams to describe system models of the component model's metamodel[6].

A system model, as a model of layer M1 in the four-layer modelling architecture, can be described in UML in two contexts (see Figure 6 and Figure 7, respectively):

1. as an object diagram where objects in M1 layer are instances of classes from a metamodel in layer M2;

[6]The aim of the utilisation is to reuse the well-established UML notation, although it is not formally defined as an UML profile.

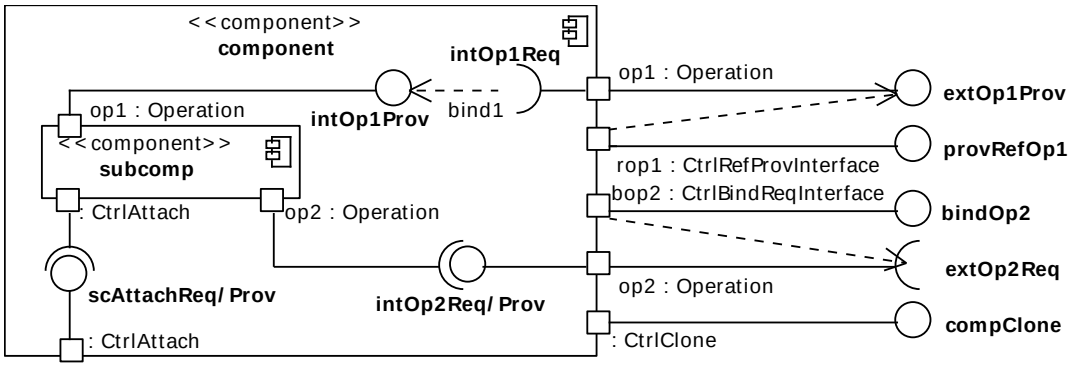

Figure 7: An example of proposed notation of a system model in layer M1 by means of the component model from layer M0.

2. as a specific class diagram where entities in layer M0 are instances of classes from a diagram in layer M1.

The following example of a simple system model is described in both contexts, i.e. as the object diagram, to demonstrate an application of the metamodel, and as a specific component diagram, to introduce the notation based on UML 2 component diagrams. The example contains composite component `component` and its primitive subcomponent `subcomp`.

The *subcomponent from the example* has one provided interface `intOp1Prov` with type `op1 : Operation`, one required interface `intOp2Req` with type `op2 : Operation`, and one required interface `scAttachReq` with a type represented by a nameless instance of class `CtrlAttach`.

The *component from the example* contains three internal interfaces, namely required `intOp1Req` with type `op1 : Operation`, provided `intOp2Prov` with type `op2 : Operation`, and provided `scAttachProv` with a type represented by a nameless instance of class `CtrlAttach`. These internal interfaces are bound to the interfaces of the subcomponent by binding `bind1` and two nameless bindings with a type represented by a nameless instance of class `BindSiblings`. Moreover, the component has five external interfaces, namely provided `extOp1Prov` with type `op1 : Operation`, required `extOp2Req` with type `op2 : Operation`, provided `compClone` with a type represented by a nameless instance of class `CtrlClone`, provided `bindOp2` with type `bop2 : CtrlBindReqInterface` binding required interface `extOp2Req`, and provided `provRefOp1` with type `rop1 : CtrlRefProvInterface` referencing provided interface `extOp1Prov`. External interfaces `extOp1Prov` and `extOp2Req` are bound to internal interfaces `intOp1Req` and `intOp2Prov` by two bindings with types represented by nameless instances of classes `BindInward` and `BindOutward`, respectively.

The example is described as an object diagram in Figure 6. All objects are either identified or nameless instances of relevant classes of the metamodel.

To provide straightforward description of the component model, as a class diagram with classes in layer M1, we utilise the notation of component diagrams from UML 2, as it is described in Figure 7. In this way, individual instances of classes from the metamodel can be denoted as follows:

- `PrimitiveComponent` – Primitive components are denoted by UML components, i.e. classes stereotyped as «`component`» (e.g. component `subcomp` in Figure 7).
- `CompositeComponent` – Composite components are denoted by UML components that are able to have subcomponents, i.e. nested UML components (e.g. component `component` in Figure 7).
- `ProvidedInterface` – Provided interfaces are denoted by UML interfaces, i.e. classes stereotyped as «`interface`» realised by related components that own the interfaces (e.g. interface `intOp1Prov` of component `subcomp` in Figure 7).
- `RequiredInterface` – Required interfaces are denoted by UML interfaces used by related components that own the interfaces (e.g. interface `intOp1Req` of component `component` in Figure 7).
- `TypeOfInterface` – A type of a component's interface is denoted by an UML port. The UML port's (optional) name and (mandatory) type are identical to a name and a class of the type of the component's interface (e.g. port `op1 : Operation` realised by interface `intOp1Prov` of component `subcomp` in Figure 7 assigns instance `op1` of class `Operation` to the interface as its type).
- `Binding` and `TypeOfBinding` – Bindings of functional required and provided interfaces are denoted by UML relations of dependency stereotyped as «`use`». Each binding can have its (optional) name and its type, if needed (e.g. binding `bind1` of required interface `intOp1Req` of component `component` and provided interface `intOp1Prov` of component `subcomp` in Figure 7). In a case of a nameless binding of interfaces, which is common for control interfaces, it is possible to interconnect the interfaces directly (e.g. the binding of interfaces `scAttachReq/Prov` of components `subcomp` and `component`, respectively, in Figure 7).
- `CtrlBindReqInterface` and `CtrlRefProvInterface` – Relations of UML ports of types `CtrlBindReqInterface` or `CtrlRefProvInterface`, which represent control provided interfaces for binding of required functional interfaces or referencing functional provided interfaces, respectively, are denoted by UML relations stereotyped as «`use`» (e.g. port `rop1 : CtrlRefProvInterface` realised by interface

provRefOp1 of component component in Figure 7 provides references to interface extOp1Prov of the same component; analogously for the port of interface bindOp2).

3.3 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 π-calculus (see Section 3.1).

According to the metamodel from Section 3.2.1, each component of a component-based system 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 π-calculus process (a value of attribute behaviouralDescription in an instance of class PrimitiveComponent, see Figure 3 in Section 3.2.1). The π-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 component-based system; the component is „a grey-box“). Formal description of the composite component's behaviour is a π-calculus process, which is composition of processes representing behaviour of the component's subcomponents, processes implementing 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.).

3.3.1 Notation

Before π-calculus processes describing behaviour of a component will be presented, we need to define the *component's interfaces* within the terms of the π-calculus, i.e. as names used by the π-calculus processes. The following names can be used in an external view or an internal view of a component, i.e. for description of an abstract component or a composite component as specific instances of classes Component or CompositeComponent, respectively:

- the external view of an abstract component: $s_0, s_1, c, p_1^s, \ldots, p_n^s, p_1^g, \ldots, p_m^g$;
- the internal view of a composite component only: $a, p_1'^s, \ldots, p_m'^s, p_1'^g, \ldots, p_n'^g$.

where 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:

via s_0 – a running component accepts a request for its stopping[7] (an interface of type CtrlStop in the metamodel),

via s_1 – a stopped component accepts a request for its starting[7] (an interface of type CtrlStart in the metamodel),

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 in the metamodel),

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 in the metamodel),

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 CtrlRefProvInterface 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 in the metamodel).

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 BindOutward in Figure 4 in Section 3.2.1).

3.3.2 Interface's References and Binding

At first, we define an auxiliary process $Wire$[8], 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).

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

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

[8] The process will be used also in the following parts of Section 3.3.

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:

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

where names $r_1,\ldots,r_n,p_1^s,\ldots,p_n^s,p_1,\ldots,p_m,p_1^g,\ldots,p_m^g$ have been defined in Section 3.3.1. 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 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 $Ctrl_{EI}$:

$$
\begin{aligned}
Ctrl_{EI} &\stackrel{def}{=} (r_1,\ldots,r_n,p_1,\ldots,p_m,r_1',\ldots,r_m',p_1',\ldots,p_n').\\
&\quad \prod_{i=1}^{n}(d)Wire\lfloor r_i,p_i',d\rfloor \mid \prod_{j=1}^{m}(d)Wire\lfloor r_j',p_j,d\rfloor
\end{aligned}
$$

3.3.3 Control of a Component's Life-cycle

Control of a composite component's life-cycle[9] can be described as process $Ctrl_{SS}$.

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

where names s_0 and s_1 represent the component's interfaces that accept stop and start requests, respectively (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` in the metamodel).

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 process $Ctrl_{SS}$ 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 subcomponent's previously attached stop and start interfaces.

3.3.4 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

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

a component as follows:

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

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

The fresh copy of a component can be used to replace a subcomponent of a composite component. The process of update[10], 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 $Update$:

$$
\begin{aligned}
Update \quad \triangleq \quad & (u, a, s_0, s_d, p_1^s, \ldots, p_m^s, p_1^g, \ldots, p_n^g).(k, s_d') \\
& (\overline{u}\langle k\rangle.k(s_0', s_1', c, r', p').\overline{s_0}.\overline{a}\langle s_0', s_1', s_d'\rangle.\overline{s_d} \\
& .r'(p_1'^s, \ldots, p_n'^s).(x)(\overline{p_1^g}\langle x\rangle.x(p).\overline{p_1'^s}\langle p\rangle \ldots \overline{p_n^g}\langle x\rangle.x(p).\overline{p_n'^s}\langle p\rangle) \\
& .p'(p_1'^g, \ldots, p_m'^g).(x)(\overline{p_1'^g}\langle x\rangle.x(p).\overline{p_1^s}\langle p\rangle \ldots \overline{p_n'^g}\langle x\rangle.x(p).\overline{p_m^s}\langle p\rangle) \\
& .\overline{s_1'}.Update\lfloor u, a, s_0', s_d', p_1^s, \ldots, p_m^s, p_1^g, \ldots, p_n^g\rfloor)
\end{aligned}
$$

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.

3.3.5 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 $s_0, s_1, r_1, \ldots, r_n, p_1, \ldots, p_m$ 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, let us assume that process abstraction $Comp_{subcomps}$ with parameters $a, p_1'^s, \ldots, p_m'^s, p_1'^g, \ldots, p_n'^g$ describes behaviour of a system of subcomponents interconnected by means of their interfaces into a composite component (see Section 3.3.2). Names $s_0, s_1, r_1, \ldots, r_n, p_1, \ldots, p_m$ and names $a, p_1^s, \ldots, p_m^s, p_1^g, \ldots, p_n^g$ are defined in Section 3.3.1.

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

$$
\begin{aligned}
Comp_{prim} \quad \stackrel{def}{=} \quad & (s_0, s_1, c, p_1^s, \ldots, p_n^s, p_1^g, \ldots, p_m^g).(r_1, \ldots, r_n, p_1, \ldots, p_m) \\
& (Ctrl_{Ifs}\langle r_1, \ldots, r_n, p_1^s, \ldots, p_n^s, p_1, \ldots, p_m, p_1^g, \ldots, p_m^g\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\
& \mid Comp_{impl}\langle s_0, s_1, r_1, \ldots, r_n, p_1, \ldots, p_m\rangle) \\
Comp_{comp} \quad \stackrel{def}{=} \quad & (s_0, s_1, c, p_1^s, \ldots, p_n^s, p_1^g, \ldots, p_m^g). \\
& (a, r_1, \ldots, r_n, p_1, \ldots, p_m, r_1', \ldots, r_m', p_1', \ldots, p_n') \\
& (Ctrl_{Ifs}\langle r_1, \ldots, r_n, p_1^s, \ldots, p_n^s, p_1, \ldots, p_m, p_1^g, \ldots, p_m^g\rangle \\
& \mid Ctrl_{Ifs}\langle r_1', \ldots, r_m', p_1'^s, \ldots, p_m'^s, p_1', \ldots, p_n', p_1'^g, \ldots, p_n'^g\rangle \\
& \mid Ctrl_{EI}\langle r_1, \ldots, r_n, p_1, \ldots, p_m, r_1', \ldots, r_m', p_1', \ldots, p_n'\rangle \mid Ctrl_{clone}\lfloor c\rfloor \\
& \mid Ctrl_{SS}\langle s_0, s_1, a\rangle \mid Comp_{subcomps}\langle a, p_1'^s, \ldots, p_m'^s, p_1'^g, \ldots, p_n'^g\rangle)
\end{aligned}
$$

where the pseudo-applications of $Ctrl_{Ifs}$ represent behaviour of control parts of the components related to their functional interfaces (see Section 3.3.2), the constant applications of $Ctrl_{clone}$ describe behaviour of control parts

[10]The process is also known as „updating“ or „nesting“ of a component.

of the components related to their cloning (see Section 3.3.4), the pseudo-application of $Ctrl_{SS}$ represents behaviour of the composite component's control part processing its stop and start requests (see Section 3.3.3), and the pseudo-application of $Ctrl_{EI}$ describes communication between internal and external functional interfaces of the composite component (see Section 3.3.2).

3.4 Behavioural Modelling of Services

This section deals with linking individual services of service-oriented architecture (SOA) to their underlying implementations as component-based systems. It provides an approach to formal description of these services as the component-based systems by means of the component model from Section 3.

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 [15]).
2. The service can be implemented as *a component-based system* (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 component 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 view shows the service as a component of CBS. Its internal structure and behaviour can be specified in the common way, as it has been described in Section 3.

3.4.1 Service as a Part of Service-Oriented Architecture

The result of business-to-service transformation [41], 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 Section 3.4). Behaviour of the service can be described as π-calculus process abstraction $Service$ as follows:

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

where names $i_1, \ldots, i_n$ 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 $j \in \{1, \ldots, n\}$, describes behaviour of processing of requests via the service's interface represented by name i_j including possible communication via shared names $b_1, \ldots, b_m$.

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 π-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 [15]. To support the dynamic service discovery and invocation, SOA provides service brokers (e.g. UDDI registries, see [12]), 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. as references to the providers' published interfaces. Its behaviour can be described as π-calculus process abstraction $Broker$ as follows:

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

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 $\overline{t}\langle t', i, d\rangle$, 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.

3.4.2 Service as a Component-Based System

A service's underlying implementation, its behaviour, and internal structure, can be described as a component-based system. 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 Section 3.4.

To describe a service `Service` with interfaces `I1` to `In` as a component-based system and by means of the component model from this thesis (see Section 3), we need to transform π-calculus process abstraction $Service$ from Section 3.4.1 describing behaviour of the service into a formal description of behaviour of a component representing the component-based system (see Section 3.3). 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

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

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 Section 3.4.1).

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 Section 3.1). 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 Section 3.3).

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

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

where process constant $Wire$ has been defined in Section 3.3.2 and process abstraction $Comp_j$ describes the component's complete behaviour and is fully compatible with behavioural description of primitive and composite components from Section 3.3.5.

3.5 Development Process

In this section, we propose an application of the component model from Section 3 and the behavioural modelling of services from Section 3.4 in a software development process. The development process should conform to principles of the component-based development (CBD) and should incorporate service-oriented architecture (SOA).

3.5.1 Application of the Behavioural Modelling of Services

When a specification of a software system is finished, we are ready to design the system's architecture. Let us assume that we have decided to use SOA and we need a formal description of behaviour of its services. We will proceed as follows:

1. Business processes of the services will be analysed at the level of individual actions and the services' goals will be modelled. The goals will be arranged into candidate services, i.e. business services encapsulating distinct sets of business logic will be identified.
2. The candidate services and their interfaces[11] will be modelled as UML classes. The classes of services will be stereotyped as «`service`», while the interfaces will be stereotyped as «`interface`» and connected to the classes of services by means of UML relationships of „realisation" (a service realises its interface, i.e. it is a „provider"). Utility services and controller services will be created and modelled in the same way as the business services. Then, the resulting UML class diagram will be transformed into an UML component diagram, according to [41], where possible interconnections of the services will be modelled. Moreover, these interconnections will be connected to the classes of services in the UML class diagram by means of UML relationships of „use" (a service uses another service via its interface, i.e. it is a „consumer"). Choreography of the services will be described in UML interaction diagrams, e.g. modelled in UML sequence diagrams.
3. Behaviour of each service will be described formally as a π-calculus process abstraction with parameters matching the service's interfaces. The process abstraction will describe complete communication behaviour of the service and will be defined directly by a developer of the service (i.e. the service is a „black-box") or by means of a specific pseudo-application of a process abstraction that will represent behaviour of an underlying component-based system (i.e. the service is a „grey-box"). In a case of services that can be published and found via service brokers and subsequently bound at runtime, behaviour of the service brokers will be also described as specific process abstractions with two parameters (i.e. „publish" and „find").

The result of the behavioural modelling of services will be a single π-calculus process abstraction representing behaviour of a whole service-oriented architecture of the system (i.e. behaviour of its main control service that orchestrates the system's private services and provides functionality publicly offered by the system, including behaviour of the private services). The behavioural description will continue with specification of behaviour of underlying component-based systems that implement individual services of the architecture (see Section 3.5.2).

3.5.2 Application of the Component Model

A component-based system can be modelled in a logical view (structural description) and in a process view (behavioural description). During the development process, we will proceed from the logical view to the process view as follows:

1. In the logical view, the component-based system will be described as a hierarchy of components obtained by applying top-down decomposition. Internal nodes of the hierarchic tree will be composite components, while its leaf nodes will be primitive components. Then, the whole component-based system will be represented as a single composite component in the tree's root. The components, at each level of the hierarchy, will have defined their provided and required interfaces of specific types. The interfaces of cooperating components will be bound (a required interface to a provided interface of an identical type). Each component will offer a set of standard control provided interfaces. Moreover, for each functional interface of the component, there will be defined auxiliary control interfaces that allow referencing (for provided) and binding (for required) the functional interface. Composite components will have additional internal functional and control interfaces that will be accessible by their direct subcomponents only. The components, their interfaces and their bindings will be described in a system model.
2. In the process view, behaviour of each primitive component will be described formally as a π-calculus process abstraction with parameters matching the component's interfaces from the logical view of the component-based system. The component's core behaviour[12] will be defined directly by a developer of the component, while its complete behaviour will be composed of the core behaviour and default behaviour of control parts that handle the component's control provided interfaces. Process abstractions and constants representing the default behaviour of the component's control parts will be adopted from the component model.
3. Finally, behaviour of each composite component will be described formally as a π-calculus process abstraction. It will be defined as a composition of process abstractions that describe behaviour of the component's subcomponents and behaviour of bindings of their interfaces in the logical view and process abstractions and constants that represent default behaviour of the component's control parts. The description of behaviour of the bindings and the component's control parts will be adopted from the component model, while the behaviour of the component's subcomponents will have been described previously. This step can be done automatically, i.e. without additional input from a developer, by processing the component hierarchy in a bottom-up approach.

[11] Services that invoke asynchronously other services have to implement extra auxiliary interfaces for receiving asynchronous replies.

[12] The core behaviour describes processing of functional interfaces and custom control interfaces only (see process abstraction $Comp_{impl}$ in Section 3.3.5).

The result of the component model's application will be the system model describing the component-based system in the logical view and a single π-calculus process abstraction representing the system's behaviour in the process view. In a case of multiple services (from Section 3.5.1) realised as underlying component-based systems, each service will be modelled and its behaviour described separately as a stand-alone component-based system.

3.5.3 Integration of a Formal Description

Formal description of behaviour of service-oriented architectures and component-based systems can be utilised in various phases of the development process, when an exact specification of a system's functionality and evolution of its architecture is needed. However, the behavioural description is probably the most useful in a design phase where it allows *to detect design faults* and *to prevent future errors* before post-design phases of the development process will be realised. For example, it is possible:

- to design and to describe the exact behaviour of systems and evolution of their architectures,
- to check if individual services and components behave correctly, they are always ready to handle external requests from their environment, and a system can not reach a forbidden state (e.g. a deadlock state or a wrong configuration of the system's architecture),
- to verify that two substitutable services or components behave equally, e.g. π-calculus processes describing their behaviour are open bisimilar,
- to validate all possible deployments of services in a service-oriented architecture or components in a component-based system's dynamic architecture, etc.

Since the formal description of behaviour uses the polyadic π-calculus without any special extensions, several existing tools can be used for model checking of resulting π-calculus processes and formal verification of their properties. The verification tools include: *The Mobility Workbench* (MWB, [45]) that is a model checker and an open bisimulation checker of mobile concurrent systems described in the π-calculus; *Another/Advanced Bisimulation Checker* (ABC, [8]) that allows to check open-equivalences in the π-calculus, in a similar way as MWB but with some improvements; and *Pi-Calculus Equivalences Tester* (PIET, [31]) that is able to check non-open equivalences, such as (strong/weak) early and late equivalence, (strong/weak) early and late congruence, and (strong/weak) ground equivalence.

4 Main Contributions

In the previous section, we have introduced the component model with support of mobile architectue and our approach to behavioural modelling of component-based systems and service-oriented architectures. The proposed component model and the behavioural modelling have been successfully validated in a case study of a service-oriented architecture for functional testing of complex safety-critical systems. The case study is available in the full version of this thesis.

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

1. *formal approaches to modelling of service-oriented architectures*, mostly based on the formalisation of business process models (e.g. transformations of BPEL to Petri nets [19] or to π-calculus processes [25, 47]);
2. *formal approaches to modelling of component-based systems*, such as component models and architecture description languages mentioned in Section1.2, which are usually focused only on CBSs without consideration of SOA at the higher level of abstraction (e.g. Wright [1], Tracta [17], behaviour protocols of SOFA [46], formal descriptions of Fractive [7], and, partially, SOFA 2.0 [11]).

Our approach intends to bridge the gap and to provide a formal description of service-oriented architectures from the choreography of their services to the behaviour of individual components of underlying component-based systems, as it has been demonstrated in the case study. Similar efforts can be found in SOFA 2.0 and the Reo coordination language [14].

In the SOFA 2.0, SOA becomes a specific case of a component-based system 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, 14] is based on the π-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 π-calculus process. Moreover, we use the polyadic π-calculus without any special extensions, which allows us to utilise existing tools for model checking of π-calculus processes and verification of their properties.

Important Merits: 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 component 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 π-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 π-calculus* without any special extensions, which allows us to utilise existing tools for model checking of π-calculus processes and formal verification of their properties.
- The proposed *behavioural modelling of service-oriented architectures* allows a developer to go from a high level service design to a more precise design of underlying component-based systems, with respect to differences between services and components. Behaviour of a whole system (individual services, their choreography, and their implementation as the underlying component-based systems) can be described as a single π-calculus process.

Possible Drawbacks: The proposed approach can suffer from the following possible drawbacks.

- The behavioural description of services and components in π-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 of services and components *requires an advanced knowledge of the π-calculus* and may be a difficult task for unskilled developers.
- This thesis describes how to model a specific configuration and behaviour of a component-based system or a service-oriented architecture as a π-calculus process. However, 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. For example, it may be difficult to determine a deadlock configuration, which has been detected by means of a verification tool in a specific π-calculus process.

5 Conclusions

In the thesis, the approach for modelling of component-based systems has been proposed. The thesis meets the objectives set out in Section 2.2. We have presented the component model, which allows to describe component-based systems with support of 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 service-oriented architectures as π-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 π-calculus process.

We have outlined the integration of our approach into a development process. In the full version of the thesis, we have illustrated an application of our approach in the case study of the environment for functional testing of complex safety-critical systems, which has been described as a service-oriented architecture and an underlying component-based system. The component-based system has been modelled as a system model based on the component model's metamodel. We have formally described behaviour of the whole environment by means of the π-calculus on the levels of the service-oriented architecture and the component-based system. 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 service-oriented architectures and transition to component-based systems, and in utilisation of the standard polyadic π-calculus, which is 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.

References

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

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

[3] ArchWare project. `http://www.arch-ware.org/`, November 2006.

[4] Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, Upper Saddle River, NJ, USA, second edition, July 2005.

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

[6] Dharini Balasubramaniam, Ron Morrison, Flavio Oquendo, Ian Robertson, and Brian Warboys. Second release of ArchWare ADL. Technical Report D1.7b (and D1.1b), ArchWare Project IST-2001-32360, June 2005.
[7] Tomás Barros. *Formal specification and verification of distributed component systems*. PhD thesis, Université de Nice – INRIA Sophia Antipolis, November 2005.
[8] Sébastien Briais. *The ABC User's Guide*, May 2005.
[9] Eric Bruneton, Thierry Coupaye, and Jean-Bernard 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.
[10] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium, February 2004.
[11] Tomáš Bureš, Petr Hnětynka, and František Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA 2006*, pages 40–48, Seattle, USA, August 2006. IEEE Computer Society.
[12] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI version 3.0.2. Uddi spec technical committee draft, OASIS Open, October 2004.
[13] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *International Conference on Software Engineering Advances, ICSEA'06*, Tahiti, French Polynesia, October 2006. IEEE.
[14] Nikolay K. Diakov and Farhad Arbab. Compositional construction of Web Services using Reo. In Savitri Bevinakoppa and Jiankun Hu, editors, *Proc. of International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI 2004)*, pages 49–58. INSTICC Press, April 2004.
[15] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, August 2005.
[16] David Garlan, Robert T. Monroe, and David Wile. ACME: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–68. Cambridge University Press, New York, NY, 2000.
[17] Dimitra Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology and Medicine University of London, Department of Computing, January 1999.
[18] Denis Hatebur, Maritta Heisel, and Jeanine 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.
[19] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets.
[20] Petr Hnětynka and František 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.
[21] 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, September 2000.
[22] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.
[23] Kung-Kiu Lau and Zheng 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.
[24] Kung-Kiu Lau and Zheng Wang. A survey of software component models (second edition). Pre-print CSPP-38, School of Computer Science, The University of Manchester, Manchester M13 9PL, UK, May 2006.
[25] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.
[26] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.
[27] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.
[28] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
[29] Vladimír Mencl. Component definition language. Master's thesis, Charles University, Prague, May 1998.
[30] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:41–77, September 1992.
[31] Matteo Mio. PiET – pi calculus equivalences tester. `http://piet.sourceforge.net/`, September 2006.
[32] OMG IDL syntax and semantics. Document formal/98-02-08, The Object Management Group, February 1998. Chapter 3 of CORBA 2.2 specification, also available as ISO/IEC 14750:1999 standard.
[33] Meta object facility (MOF) specification, version 1.4.1. Document formal/05-05-05, The Object Management Group, July 2005. Also available as ISO/IEC 19502:2005 standard.

[34] Unified modeling language specification, version 1.4.2. Document formal/05-04-01, The Object Management Group, January 2005. Also available as ISO/IEC 19501:2005 standard.
[35] Meta object facility (MOF) core specification, version 2.0. Document formal/06-01-01, The Object Management Group, January 2006.
[36] UML infrastructure, version 2.1.2. Document formal/2007-11-04, The Object Management Group, November 2007.
[37] UML superstructure, version 2.1.2. Document formal/2007-11-02, The Object Management Group, November 2007.
[38] Flavio Oquendo. π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29:1–14, 2004.
[39] SCA service component architecture: Assembly model specification. Technical Report SCA version 1.00, Open SOA Collaboration, March 2007.
[40] František Plášil, Dušan Bílek, and Radovan Janeček. SOFA/DCUP: Architecture for component trading and dynamic updating. In *4th International Conference on Configurable Distributed Systems*, pages 43–51, Los Alamitos, CA, USA, May 1998. IEEE Computer Society.
[41] Marek Rychlý and Petr 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, May 2008.
[42] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New Ed edition, October 2003.
[43] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, second edition, November 2002.
[44] Steve Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, February 1993.
[45] Björn Victor. *The Mobility Workbench User's Guide*, polyadic version 3.122 edition, October 1995.
[46] Stanislav Višňovský. *Modeling software components using behavior protocols*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002.
[47] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient analysis of BPEL 2.0 processes using π-calculus. In *APSCC '07: Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference*, pages 266–274, Washington, DC, USA, 2007. IEEE Computer Society.