

Marek Rychlý, Jaroslav Zendulka

Modelling of Component-Based Systems with Mobile Architecture

Monograph

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Editorial board of Faculty of Information Technology:

Prof. Jaroslav Zendulka
Department of Information Systems
chair

Prof. Tomáš Hruška
Department of Information Systems

Adam Herout, Ph.D.
Department of Computer Graphics and Multimedia

Prof. Milan Češka
Department of Intelligent Systems

Prof. Alexander Meduna
Department of Information Systems

Lukáš Sekanina, Ph.D.
Department of Computer Systems

Petra Nastulczyková
Library

© 2010 Faculty of Information Technology, Brno University of Technology
Monograph

Cover design 2010 by Dagmar Hejduková

Published by Faculty of Information Technology,
Brno University of Technology, Brno, Czech Republic

Printed by MJ servis, spol. s r.o.

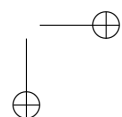
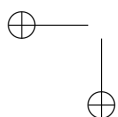
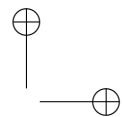
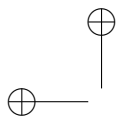
ISBN 978-80-214-4211-5

Preface

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 this book, 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. It is shown that the component model addresses the dynamic aspects of software architectures including the component mobility. Furthermore, a method of behavioural modelling of service-oriented architectures is proposed 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 the 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 the approach.

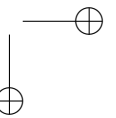
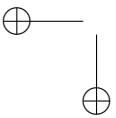
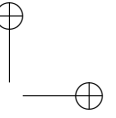
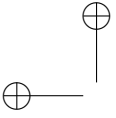


Acknowledgment

I would like to thank all the team of the Department of Information Systems, at BUT FIT, for their valuable suggestions regarding this work. Special thanks belong to my family for their patience, encouragement, and support over the years.

Marek Rychlý
Brno, 2010

The work presented in this book has been supported by the long-term institutional research project of the Czech Ministry of Education number CEZ MSM 0021630528 „Security-Oriented Research in Information Technology“, the BUT FIT grant FIT-10-S-2 „Recognition and presentation of multimedia data“, and the Grant Agency of the Czech Republic grant number 102/05/0723 „A Framework for Formal Specifications and Prototyping of Information System’s Network Applications“.



Contents

1	Introduction	1
1.1	Objectives of This Book	3
1.1.1	Overview of the State of the Art	4
1.1.2	Component Model for Mobile Architectures	4
1.1.3	Application of the Component Model in SOA	5
1.1.4	Software Development Process	5
1.1.5	Case Study	5
1.2	Structure of This Book	5
<hr/>		
Part I State of the Art		
<hr/>		
2	Formal Bases	9
2.1	Labelled Transition Systems	10
2.2	Communicating Sequential Processes	11
2.3	Calculus of Mobile Processes	13
2.3.1	Operational Semantics	16
2.3.2	Congruences of Processes	17
3	Software Component Architecture	21
3.1	Software Architecture	21
3.2	Component-Based Development	22
3.3	Component Models	23
3.3.1	Wright	23
3.3.2	Darwin and Tracta	25
3.3.3	SOFA	26
3.3.4	SOFA 2.0	27
3.3.5	Fractal	28
3.4	Architecture Description Languages	30
3.4.1	ACME	31
3.4.2	Unified Modelling Language	31

VIII Contents

3.4.3	ArchWare ADL	34
4	Service Oriented Architecture	37
4.1	Design of Services	37
4.1.1	Business Process Modelling	38
4.1.2	Business-to-Service Transformation	39
4.1.3	Service Composition	39
4.2	Implementation of Services	40
4.3	Services and Components	41
4.3.1	Service Component Architecture	41

Part II Component Model for Mobile Architectures

5	Component Model	47
5.1	Logical View	47
5.1.1	Metamodel	49
5.1.2	System Model	55
5.2	Process View	58
5.2.1	Notation	59
5.2.2	Interface’s References and Binding	60
5.2.3	Control of a Component’s Life-cycle	61
5.2.4	Cloning of Components and Updating of Subcomponents	62
5.2.5	Primitive and Composite Components	63
5.3	An Example of a Component-Based System and its Description	64
5.3.1	Definition of the Components’ Implementations	65
5.3.2	Description of the Component Based System	67
6	Behavioural Modelling of Services	69
6.1	Service as a Part of Service Oriented Architecture	70
6.1.1	Communication of Services and Service Broker	70
6.2	Service as a Component Based System	71
6.3	An Example of a Service-Oriented Architecture	73
6.3.1	Service Identification	73
6.3.2	Service Model	75
6.3.3	Description of Services as Entities of SOA	77
6.3.4	Description of Services as Component-Based Systems ..	78

Part III Application

7	Development Process	83
7.1	Application of the Behavioural Modelling of Services	83
7.2	Application of the Component Model	84
7.3	Integration of a Formal Description	86

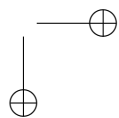
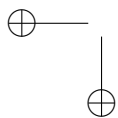
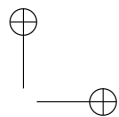
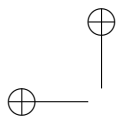
8	Tools	87
8.1	Component Modelling Tools	87
8.1.1	Component Diagrams in UML	87
8.1.2	A Tool for Modelling of Component-Based Systems	88
8.2	Verification Tools	90
8.2.1	The Mobility Workbench (MWB)	92
8.2.2	Another/Advanced Bisimulation Checker (ABC)	93
8.2.3	Pi-Calculus Equivalences Tester (PiET)	93
9	Case Study	95
9.1	System Description	96
9.2	Service Identification	97
9.3	Component-Based System	100
9.4	Formal Description of the Service-Oriented Architecture	102
9.5	Formal Description of the Component-Based System	104
9.6	System Properties and Their Verification	108
9.6.1	Simulation	108
9.6.2	Deadlocks	109
9.6.3	Bisimulation Checking	110
9.6.4	Model Checking	111
9.7	Evaluation and Conclusion	112
9.7.1	Important Merits	113
9.7.2	Possible Drawbacks	114

Part IV Conclusion

10	Summary	117
11	Future Research Directions	119
	References	121
	Acronyms	129

Part V Appendices

	Process Descriptions from the Case-Study in MWB/ABC	135
A.1	Control Parts of Components	135
A.2	Core Behaviour of the Components	136
A.3	Complete Behaviour of the Subcomponents	137
A.4	Behaviour of the Composite Component	138
A.5	Services of SOA	139



List of Figures

3.1	An example of UML „class“ style notation with interface stereotypes and corresponding „lollipop“ style notation where Borrow acts as an „assembly connector“ between Library and Book or CD (the example is adopted from [AN05]).	33
3.2	An example of component Store , its internal structure and components Order , Customer , and Product , as parts of its internal assembly (the example is adopted from [OMG07b]).	34
5.1	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).	48
5.2	A simplified part of the EMOF metamodel [OMG06a] with classes that will be extended by the component model.	49
5.3	Abstract component, realisations, and interfaces, extending EMOF::NamedElement in the metamodel of the component model.	50
5.4	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 5.3.	51
5.5	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 5.3.	52
5.6	The example of description of a system model as an object diagram with instances of classes from the component model’s metamodel.	56
5.7	An example of proposed notation of a system model in layer M1 by means of the component model from layer M0.	56

XII List of Figures

5.8 The example of a simple component-based system that dynamically changes its behaviour, component `system` and its subcomponents `init`, `workerA`, and `workerB` (an initial configuration, i.e. without bound interface `sysFunc`)..... 65

6.1 Business process model of „Process Purchase Order“ (adopted from [OMG06b])..... 74

6.2 An overview of identified services and their interconnections... 75

6.3 Controller service `Scheduling` and its orchestration of business services `ProductionScheduling` and `ShippingScheduling`..... 76

6.4 Behaviour of service `Scheduling` as a sequence of service invocations..... 77

8.1 Eclipse Ecore diagram of the metamodel, which is used in the tool for modelling of component-based systems (adopted from [Gal09], a full version can be found in [Ryc09]). 89

8.2 The model of the component-based system from the example in Section 5.3 (adopted from Figure 5.8) with component `system` and its subcomponents `init`, `workerA`, and `workerB`, without control interfaces. 89

9.1 Testing environment of a railway interlocking control system (adopted from [DMM⁺08]). 96

9.2 An overview of identified services of the testing environment and their interconnections. 97

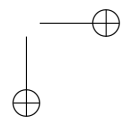
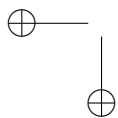
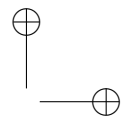
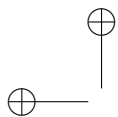
9.3 The choreography of services in the testing environment. 98

9.4 Services of the testing environment as UML classes. 99

9.5 Structure of composite component `TestEnvironment` with attached component `test`. 101

List of Tables

- 4.1 The comparison of Service Oriented Architecture (SOA) and Component-Based Development and Systems (CBD/CBS), which has been published in [Ryc08]..... 41



1

Introduction

Increasing globalisation of information society and its progression create needs for extensive and reliable information technology solutions. A few years ago, IT solutions for support of an entire organisation were, in most cases, applications of several independent and specialised information systems (an accounting system, a production system, etc.). Nowadays, complex information systems are required, which provide information support across the organisation's departments. Common features of the mentioned systems include [KŽ00, KŽ03]

- adaptability to variable structures of organisations and distributed activities – support of highly autonomous subunits and their collaboration, the ability to use critical functions of components even in the situation when the whole system does not work,
- integration of well-established software products – it implies lower costs, the ability to integrate and use legacy systems and third party products, reduction of dependence on one supplier,
- scalability and high adaptability to variable requirements – the ability to customise „general systems“ instead of building new systems „from scratch“, continuous and endless evolution of the systems together with organisations (e.g. selling of some divisions of companies, fusions of companies, changing business conditions),
- connection to a variable set of external systems (e.g. a variable set of „e-business“ partners) and systems of cooperating organisations (great projects must be often realised by a consortium of several big companies), etc.

It is obvious that the mentioned features have significant impact on architectures of software systems. The systems can not be realised as monoliths. The exact specification of functions and interfaces of the systems' parts is necessary, as well as specification of their deployment and communication. Moreover, integration of third party products often requires gateways adapting interfaces of the products to the systems' interfaces.

2 1 Introduction

Therefore, the information systems of organisations are realised as networks of quite autonomous, but cooperative, units communicating asynchronously via messages of appropriate format. Such systems [KŽ03] are called *software confederations* (SWCs, with components working as permanently available services) or *software alliances* (SWAs, semi-confederations, which are formed temporarily during the systems’ runtime).

Design and implementation of SWC/SWA have to deal with many problems including [KŽ00]

- the ability to clone components (i.e. to make their copies) and to move them across a network – e.g. to optimise the system behaviour (distributed processing),
- dynamic reconfiguration of the systems – creation and destruction of components during the systems’ runtime, updating components, maintaining components’ compatibility,
- collaboration of autonomous components – how to find components having an ability to solve a given task, how to verify that the task was finished correctly,
- programmable component interfaces – one component can have many interfaces, choice of an interface depends on required functionality, security, communication protocol, version, etc.

Moreover, there are critical applications where SWC/SWA systems are getting involved. Their architectures are evolving during the systems’ runtime and their formal specification is necessary. Design of the distributed systems with *dynamic architectures* (i.e. architectures with dynamic reconfigurations) and *mobile architectures* (i.e. dynamic architectures with component mobility) can not be done by means of conventional software design methods (e.g. UML). In most cases, these methods are able to describe semi-formally only sequential processing or simple concurrent processing bounded to one component without advanced features such as dynamic reconfiguration.

Nowadays, there are two approaches related to SWC/SWA systems: service-oriented architecture and component-based development.

The *service-oriented architecture* (SOA, [Erl05]) is a widely used architectural style for design of distributed software systems at a higher level of abstraction. It covers the whole development process from an analysis where individual services are derived from user requirements (usually represented by a system of business processes) to an implementation, which uses particular technologies implementing the services (e.g. Web Services).

The *component-based development* (CBD, [Szy02]) is a software development methodology, which is strongly oriented to composability and reusability in a software system’s architecture at a lower level of abstraction. In the CBD, from a structural point of view, a software system is composed of *components*, which are self-contained entities accessible through well-defined *interfaces*. *Component models* are specific metamodels of software architectures supporting the CBD.

Current approaches to SOA design usually end up at the level of individual services. They do not describe underlying systems of components, which form design of individual services as component-based software systems with well-defined interfaces and behaviour. Moreover, CBD has limitations in formal description, which restrict the full support of the mobile architectures. Those restrictions can be delimited by usage of formal bases, which do not consider dynamic reconfigurations and component mobility, and strict isolation of control and business logic of components that does not allow full integration of dynamic reconfigurations into the components’ behaviour.

1.1 Objectives of This Book

The aim of this book is to analyse the state of the art of modelling of component-based systems and to introduce a novel component model for description of mobile architectures (i.e. fully dynamic architectures including mobility of their entities).

The component models that are described in the state-of-the-art overview in this book (see Section 3.3) support formal description of a software architecture and behaviour of its components. Moreover, recent component models, such as SOFA 2.0 and Fractal, have introduced partial support for dynamic architectures (see Sections 3.3.4 and 3.3.5), which are also supported by recent architecture description languages (see Section 3.4.3).

However, those component models and architecture description languages have many limitations with respect to support of mobile architectures, incorporation of component-based design into service-oriented architecture and into software development processes in general. The limitations result from the following *problem factors*:

- F1: usage of formal bases or models that usually do not consider component mobility (e.g. PNETs in Fractal [Bar05], behaviour protocols in SOFA [Viš02], and reconfiguration patterns in SOFA 2.0 [HP06]; for details, see the relevant parts of Section 3.3);
- F2: strict isolation of components from their controllers, which does not allow full integration of architecture reconfiguration into behaviour of the components (e.g. restrictions of PNETs in a formal description of Fractal components [Bar05] where functional operations can not fire control operations; for details, see Section 3.3.5);
- F3: insufficient support for description of service-oriented architectures where individual services can be implemented as underlying component-based systems (e.g. in Fractal component model [BCS04] or in the ArchWare project [Arc06]; for details, see Sections 3.3.5 and 3.4.3 and Chapter 4);
- F4: inconsistency between development of component-based systems and well-established software development processes of standard software systems [HHS06], difficult modelling of the component-based systems during the

4 1 Introduction

development processes (e.g. as a consequence of different conceptions of components in the component models [LW05], in component diagrams of UML [OMG05b, OMG07b] or architectures of SCA [OSO07a]; for details, see Sections 3.3, 3.4.2, and 4.3.1);

F5: insufficient integration of description of component-based systems, formal description of their behaviour, and application of related formal methods into software development processes [BH95, BH06, Hal90] (e.g. Fractal/Fractive component model [Bar05] or the ArchWare project [Arc06] provide required formalisms and tools, but they do not integrate the formalisms and the tools into a development process; for details, see Sections 3.3.5 and 3.4.3).

To address the mentioned problem factors, this book introduces a novel component model and its formal basis supporting features of mobile architectures. The book proposes also a method for application of the component model in service-oriented architectures, to develop mapping rules between services and component-based systems described by means of the component model, and finally, it demonstrates the application of the proposed approach on a case study, to evaluate its important merits and possible drawbacks over the existing conventional approaches. The component model will be fully applicable to modelling of SWC/SWA systems, to modelling of component-based systems as well as service-oriented architectures.

The *specific objectives* of this book can be summarised as in the following sections.

1.1.1 Overview of the State of the Art

This book provides an overview of the state of the art of current component models that implement dynamic architectures, analyses architecture description languages that are suitable for description of component-based systems with dynamic architectures, and explores relevant formal bases that are able to support behavioural modelling of components in dynamic architectures. The main focus is put on advanced features of the dynamic architectures, such as dynamic update and mobility of components. Moreover, service-oriented architecture are analysed in terms of component-based development.

1.1.2 Component Model for Mobile Architectures

This book introduces a novel component model and its formal basis supporting features of mobile architectures and addressing the current issues of component-based development, e.g. it integrates functional operations and relevant behaviour of components with control operations enabling dynamic reconfiguration. The component model is described as a metamodel, which allows construction of specific models of component-based systems with mobile architectures. The models are able to describe static structure of the systems

as well as particular relations between their components and interfaces needed for dynamic reconfiguration and component mobility. Supporting formal basis for behavioural description of the component model’s entities is adapted in parallel with the description of the metamodel to ensure their maximal compatibility. This objective addresses problem factors F1 and F2.

1.1.3 Application of the Component Model in SOA

This book describes a method of application of the component model and its formal description in service-oriented architectures. To bridge a gap between individual services and component-based systems, the services can be modelled as underlying component-based software systems with well-defined interfaces and behaviour described by means of the component model. This objective deals with problem factor F3.

1.1.4 Software Development Process

This book proposes an application of the component model and the modelling of service-oriented architecture in a software development process and possible utilisation of the formal description of behaviour of services in service-oriented architectures and components in component-based system. This objective includes introduction of modelling and verification tools and is focused on problem factors F4 and F5.

1.1.5 Case Study

Finally, this book demonstrates the application of service-oriented architectures, the component model, and the behavioural description of services and underlying component-based systems on a case study. The case study deals with a service-oriented architecture for functional testing of complex safety-critical systems and it evaluates effectiveness and robustness of the component model with support of mobile architectures over the existing conventional approaches.

1.2 Structure of This Book

This book is divided into 4 parts, as follows: Part I „State of the Art“, Part II „Component Model for Mobile Architectures“, Part III „Application“, and Part IV „Conclusion“. The parts consist of Chapters 2–4, Chapters 5–6, Chapters 8–9, and Chapters 10–11, respectively.

6 1 Introduction

Part I: State of the Art

In Chapter 2, we provide formal bases, a brief introduction to process algebras, which are later used to describe component-based systems as networks of communicating processes. In Chapter 3, we define software architecture in general, describe CBD in more detail, and review component models and architecture description languages, which are relevant to our subject. Chapter 4 gives an introduction to SOA with a focus on composition and implementation of services.

Part II: Component Model for Mobile Architectures

In Chapter 5, a novel component model with support of mobile architectures and formal description is introduced, addressing the current issues of the existing component models and architecture description languages. Chapter 6 deals with behavioural modelling of service as parts of SOA and as component-based systems by means of the presented component model.

Part III: Application

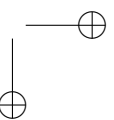
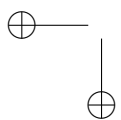
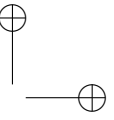
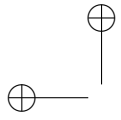
In Chapter 7, an application of the component model and the behavioural modelling of services is proposed in a software development process. In Chapter 8, an overview of tools supporting our component model is provided, including related tools for the model checking. In Chapter 9, we describe a detailed case study on the proposed approach, evaluate its results, and discuss advantages and disadvantages of the presented approach.

Part IV: Conclusion

In Chapter 10, we briefly summarise the main methods presented in this book and emphasise the most important results. To conclude, in Chapter 11, we outline future research directions in modelling of component-based systems with mobile architecture.

Part I

State of the Art



2

Formal Bases

In software engineering, formal methods are mathematically-based techniques for specification, development, and verification of software systems. Application of the formal methods aims to increase reliability and robustness of complex software systems by means of their formal description and subsequent formal verification. There are a few formal methods [CW96], which can be suitable for specifying a component-based systems' desired behavioural and structural properties. Yet, some formal methods, such as *Z notation* [ISO02], focus mainly on description of sequential systems [Eva94], while the component-based systems are (in most cases) concurrent systems. The suitable can be formal methods such as temporal logic, automata-based methods, and process algebras.

The *temporal logic* allow to describe component-based systems declaratively (e.g. an approach in [AM02]). This is useful for specifying (restricting) a system's properties, but does not allow to describe activities and generate an executable model of the system, for example. Therefore, the temporal logic are used in combination with other approaches.

The automata-based methods and process algebras describe component-based systems imperatively – behaviour is defined in terms of sequences and synchronisations of actions. The *automata-based methods* define finite transition systems with input, output, and internal actions. This allows direct application of a wide range of well-known formal algorithms, but provides only general low-level abstraction where advanced features must be implemented explicitly. As examples of the automata-based methods, we can mention *Interface automata* [dAH01] and *Component-interaction automata language* [ČVZ06].

The *process algebras* regard component-based systems as networks of communicating processes, providing high-level abstractions that can include advanced features of dynamic and mobile architectures. The processes are represented by objects in some mathematical domain and the systems' behaviour described by means of applications of operators within an algebraic theory. Moreover, the systems given in the algebras can be related by their behaviour

via equivalences and preorders, which allows reasoning about such systems through the relations.

In this chapter, we start by introducing the process algebra theories, which give the formal bases of current approaches in Chapter 3 and also of our approach in Part II of this book. At first, in Section 2.1, we introduce labelled transition systems (LTSs) as models for implementation of the process algebras’ operational semantics. We describe two process algebras. In Section 2.2, we briefly introduce communicating sequential processes (CSP). The second process algebra is *calculus of mobile processes* described in more detail in Section 2.3. We focus mainly on the calculus of mobile processes, which is intensively used as a formal basis to the original approach proposed in this book.

2.1 Labelled Transition Systems

A *state transition system*, or simply a *transition system* (TS), is an abstract machine describing behaviour of a system. The transition system consists of a set of the system’s states and transitions between these states¹.

Definition 1 (Transition System) *A transition system is a pair $\mathcal{A} = (S, \rightarrow)$ where S is a finite or infinite set of states and $\rightarrow \subseteq S \times S$ is a finite or infinite set of transitions (a transition relation) between the states. For transition $(s, t) \in \rightarrow$ where $s, t \in S$, which we can write as $s \rightarrow t$, state s is a source state and state t is a target state of the transition.*

The *labelled transition system* (LTS) is a transition system where each its transition has assigned a label². Those labels represent actions or events, which trigger the transitions.

Definition 2 (Labelled Transition System) *A labelled transition system is a triple $\mathcal{A} = (S, L, \rightarrow)$ where S is a finite or infinite set of states, L is a finite or infinite set of labels (an alphabet), and $\rightarrow \subseteq S \times L \times S$ is a finite or infinite set of transitions (a transition relation) between the states by means of the labels. For transition $(s, l, t) \in \rightarrow$ where $s, t \in S$ and $l \in L$, which we can write as $s \xrightarrow{l} t$, state s is a source state and state t is a target state of the transition.*

The sets of states and transitions need not to be necessarily finite, or even countable. Transition systems with a finite number of states and transitions

¹ We do not consider an initial state, because TS describes only the system’s (observable) behaviour, not its start.

² There exist many various formal definitions of LTS, e.g. in [Bar05] the transition relation is replaced by two functions from L to S (each maps a label into a source and target state of a transition).

2.2 Communicating Sequential Processes 11

can be represented as directed graphs (nodes are states and edges are transitions). A TS or a LTS is deterministic, iff its transition relation \rightarrow is a really partial function from S to S or $S \times L$ to S , respectively.

According to [Bar05], we can define a synchronisation constraint, a synchronisation vector, and a synchronous product as follows.

Definition 3 (Synchronisation Constraint) A synchronisation constraint of sets of labels L_1, \dots, L_n is a subset $I \subseteq L_1 \times \dots \times L_n$.

Definition 4 (Synchronisation Vector) A synchronisation vector is an element $v \in I$ of a synchronisation constraint I .

Definition 5 (Synchronisation Product) A synchronous product of n LTSs $(S_n, L_n, \rightarrow_n)$ under a synchronisation constraint I is a LTS (S, L, \rightarrow) where $S = S_1 \times \dots \times S_n$, $L = L_1 \times \dots \times L_n$, and $\rightarrow = I \cap (\rightarrow_1 \times \dots \times \rightarrow_n)$.

The synchronisation product of LTSs under a synchronisation constraint has been introduced for purpose of hierarchical composition of the LTSs. It defines a system of the LTSs where each synchronisation vector of the system’s synchronisation constraint represents the system’s global transition, i.e. a group of concurrent transitions over the system’s LTSs. In other words, the synchronisation constraint defines global transitions that are visible in the synchronous product, as transactions of the system. It also allows to hide some global transitions as „internal“.

The LTS formalism is used as a formal basis for Tracta, which defines formal semantics of the component model Darwin (see Section 3.3.2), and a PLTS/PNET formalism used for a formal description of systems in the Fractal component model (see Section 3.3.5). Generally, labelled transition systems are used to describe operational semantics of process algebras. In this book, we will use labelled transition systems to describe operational semantics of π -calculus in Section 2.3.1.

2.2 Communicating Sequential Processes

The process calculus of *Communicating Sequential Processes* (CSP, see [Ros98]) is a formal language for describing patterns of interaction in concurrent systems with static structure. The CSP was introduced by Charles Hoare in 1978 and has evolved substantially over the years. It provides communication *events* and *processes*:

- a communication event – an input/output event, a name from an alphabet which contains all possible communications for processes in the universe under consideration;
- a process – it represents a fundamental behaviour and is able to interact with other process solely through message-passing communication.

Processes may have to co-operate in the performance of an event, which happens only when all its participants³ are prepared to execute it (known as „handshaken communication“) and inevitably at the moment when these participants have agreed to execute it (the event is „instantaneous“).

In CSP, there are two *special processes*: STOP that does not communicate (also called a „deadlock“) and SKIP that represents successful termination.

Formally, the CSP’s semantics has been defined in [Ros98] as an *operational semantics*, by means of labelled transition systems described in Section 2.1. However, we can describe the CSP informally by introducing its basic *algebraic operators*:

- $\alpha \rightarrow P$ – a *prefix operator*, the process is initially willing to communicate event α and will wait indefinitely for this α to happen, after α it behaves like process P ;
- $(\alpha \rightarrow P) \square (\beta \rightarrow Q)$ – a *deterministic „external“ choice* (iff $\alpha \neq \beta$), the environment of process can choose any one of the events α or β and the subsequent behaviour will be the corresponding process P or Q , respectively;
- $(\alpha \rightarrow P) \sqcap (\beta \rightarrow Q)$ – a *non-deterministic „internal“ choice* (iff $\alpha = \beta$);
- $P \parallel \{\alpha\} \parallel Q$ – an *interface parallel operator*, which represents concurrent activity synchronised via event α (the interface);
- $P \parallel \parallel Q$ – an *interleaving operator* for independent concurrent activities;
- $(\alpha \rightarrow P) \setminus \{\alpha\}$ – a *hiding operator* making event α unobservable.

To apply the prefix operator to a set of events, CSP defines a „*prefix choice construct*“, $?x : A \rightarrow P(x)$. It allows a process $P(x)$ for each $x \in A$, where $A = \{a_1, \dots, a_n\} \subseteq \Sigma$ is any set of events, to accept any element $a \in A$ and then behave like the appropriate process $P(a)$.

$$?x : A \rightarrow P(x) \stackrel{\text{def}}{=} (a_1 \rightarrow P(a_1)) \square \dots \square (a_n \rightarrow P(a_n))$$

The „prefix“ choice construct can be used to introduce *input and output events* that are receiving and sending given *objects* via given *channels*, respectively. Let Σ is an alphabet of events containing compound objects, which are put together by an exclamation mark „!“ , and $c!T = \{c!x \mid x \in T\} \subseteq \Sigma$. The first components of the events represent the channels and the second components are the communicated objects.

An input of value y of type T over channel c can be written in the form $?y : c!T \rightarrow P(y)$, where the uses of y in $P(y)$ have to extract x from $c!x$. However, it is more elegant to use the form

$$c?x : T \rightarrow P'(x)$$

where the definition of P' will be slightly simpler than P because it can refer to value x received along c directly rather than having to recover it from a compound object.

³ The participants can be two processes or a process and an environment.

The [Ros98] presents an example of process *COPY*, which inputs elements of T on channel *left* (i.e. $left?x : T \rightarrow \dots$) and outputs them on channel *right* (i.e. $right!x \rightarrow \dots$).

$$COPY \stackrel{\text{def}}{=} left?x : T \rightarrow right!x \rightarrow COPY$$

A present-day CSP introduces also additional binary operators *sequential composition*, *timeout* and *interrupt*, a ternary operator *conditional choice*, and another various operators. The CSP language is a formal basis of the component model WRIGHT described in Section 3.3.1.

2.3 Calculus of Mobile Processes

The process algebra π -calculus, known also as a *calculus of mobile processes* [MPW92], is an extension of Robin Milner’s *calculus of communicating systems* (CCS). This section will briefly summarise the fundamentals of the π -calculus, a theory of mobile processes, according to [SW03]. The π -calculus allows modelling of systems with dynamic communication structures (i.e. mobile processes) by means of two concepts:

- a process – an active communicating entity in a system, primitive or expressed in π -calculus (denoted by uppercase letters in expressions)⁴;
- a name – anything else, e.g. a communication link (a port), variable, constant (data), etc. (denoted by lowercase letters in expressions)⁵.

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

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

Definition 6 (π -calculus) *The processes, the summations, and the prefixes of the π -calculus are given respectively by*

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

⁴ A parametric process is also called „an agent“.

⁵ The names can be called according to their meanings (e.g. a port/link, a message, etc.).

We will give a brief, informal account of semantics of π -calculus processes. At first, process 0 is a π -calculus process that can do nothing, it is the *null process* or *inaction*. If processes P and P' are π -calculus processes, then following expressions are also π -calculus processes with formal syntax according to the Definition 6 and given informal semantics:

- $\bar{x}\langle y \rangle.P$ is an *output prefix* that can send name y via name x (i.e. via the communication link x) and continue⁶ 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 occurrence⁷ of name z in the process;
- $\tau.P$ is an *unobservable prefix* that can evolve invisibly to process P , it can do an internal (silent) action and continue as process P ;
- $P + P'$ is a *sum* of capabilities of P together with capabilities of P' processes, it proceeds as either process P or process P' , i.e. when a sum exercises one of its capabilities, the others are rendered void;
- $P \mid P'$ is a *composition* of processes P and P' , which can proceed independently and can interact via shared names;
- $(z)P$ is a *restriction* of the scope⁸ of name z in process P ;
- $!P$ is a *replication* that means an infinite composition of processes P or, equivalently, a process satisfying the equation $!P = P \mid !P$.

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

Definition 7 (Binding) *In each of $x(z).P$ and $(z)P$, the displayed occurrence of z is binding with scope P . An occurrence of a name in a process is bound if it is, or it lies within the scope of, a binding occurrence of the name, otherwise the occurrence is free.*

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

Since the sum and composition operators are associative and commutative (according to the relation of structural congruence [MPW92]) they can be used with multiple arguments, independently of their order. Also an order of application of the restriction operator is insignificant. We will use the following notations:

- for $m \geq 3$, let $\prod_{i=1}^m P_i = P_1 \mid P_2 \mid \dots \mid P_m$ be a *multi-composition* of processes P_1, \dots, P_m , which can proceed independently and can interact via shared names;

⁶ The prefix ensures that process P can not proceed until a capability of the prefix has been exercised.

⁷ See also Definition 7.

⁸ The scope of a restriction may change as a result of interaction between processes.

- for $n \geq 2$ and $\tilde{x} = (x_1, \dots, x_n)$, let $(x_1)(x_2) \dots (x_n)P = (x_1, x_2, \dots, x_n)P = (\tilde{x})P$ be a *multi-restriction* of the scope of names x_1, \dots, x_n to process P .

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

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

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

Definition 8 (Abstraction) *An abstraction of arity $n \geq 0$ is an expression of form $(x_1, \dots, x_n).P$, where the x_i are distinct. For $n = 1$, the abstraction is a monadic abstraction, otherwise it is a polyadic abstraction⁹.*

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

Definition 9 (Pseudo-application) *If $F \stackrel{def}{=} (\tilde{x}).P$ is of arity n and \tilde{y} is length n , then $P\{\tilde{y}/\tilde{x}\}$ is an instance of F . We abbreviate $P\{\tilde{y}/\tilde{x}\}$ to $F\langle\tilde{y}\rangle$. We refer to this instance operation as pseudo-application of an abstraction.*

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

Definition 10 (Constant application) *A recursive definition of a process constant K is an expression of the form $K \stackrel{\Delta}{=} (\tilde{x}).P$, where \tilde{x} contains all names that have a free occurrence in P . A constant application, sometimes referred as an instance of the process constant K , is a form of process $K[\tilde{a}]$.*

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

Definition 11 (Reduction) *The reduction relation, \rightarrow , is defined by the following rules:*

⁹ The π -calculus that uses the polyadic abstractions is known as polyadic π -calculus [SW03].

16 2 Formal Bases

$$\begin{array}{l}
 \text{R-INTER} \frac{(\bar{x}(y).P_1 + M_1) \mid (x(z).P_2 + M_2) \rightarrow P_1 \mid P_2\{y/z\}}{\text{R-TAU} \frac{\tau.P + M \rightarrow P}{} \\
 \text{R-PAR} \frac{P_1 \rightarrow P'_1 \quad P_2 \rightarrow P'_2}{P_1 \mid P_2 \rightarrow P'_1 \mid P'_2} \qquad \text{R-RES} \frac{P \rightarrow P' \quad (z)P \rightarrow (z)P'}{(z)P \rightarrow (z)P'} \\
 \text{R-STRUCT} \frac{P_1=P_2 \rightarrow P'_2=P'_1 \quad P_1 \rightarrow P'_1}{P_1 \rightarrow P'_1} \qquad \text{R-CONST} \frac{K[\tilde{a}] \rightarrow P\{\tilde{a}/\tilde{x}\}}{K \triangleq (\tilde{x}).P}
 \end{array}$$

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

The exact description of operational semantics for the π -calculus can be found in [MPW92], formally described and explained in terms of *labelled transition systems* (LTSs (see Section 2.1), and is described in Section 2.3.1.

The process algebra π -calculus is a formal basis of the component model Darwin (Section 3.3.2) and the architecture description language of the project ArchWare (Section 3.4.3). It has been influenced by first versions of the mentioned CSP language (Section 2.2) and influences development of modern CSP [Ros98]. The π -calculus supports description of systems with dynamic architectures.

2.3.1 Operational Semantics

In this section, we will introduce operational semantics of π -calculus in terms of LTSs (see Section 2.1). The calculus of mobile processes will be used later in this book as a formal basis for our approach in Section 5.2.

In π -calculus, we distinguish two ways to treat input actions: early instantiation and late instantiation. The *early instantiation* means that a variable received by a process is instantiated immediately, at the time of inferring the input action, as a new name. In the *late instantiation*, the input action does not instantiate a variable as a new name, but rather it refers to the original name, which has been sent (the variable becomes instantiated only when inferring an internal communication). The operational semantics described in this section (and in the book) uses the early instantiation.

The π -calculus processes evolve by performing free and bounded „output“ actions, „input“ actions, and „unobservable“ actions [SW03].

Definition 12 (Actions) *The actions in π -calculus are given as $\alpha \in L$ in forms*

$$\alpha ::= x(z) \mid \bar{x}(y) \mid \bar{x}[z] \mid \tau$$

The actions are identical to the prefixes in Definition 6 (see Section 2.3), except for bounded output $\bar{x}[z]$ that represents sending via x a fresh name z , which become binding with scope of a process that proceeds after sending z and a process that receives z (see Definition 7). A π -calculus process and its evolution by performing the actions L are given by a LTS where transition relations are defined [SW03] as follows.

Definition 13 (Transition relations) *The (early) transition relations, $\{\overset{\alpha}{\rightarrow} \mid \alpha \in L\}$, are defined by the following rules¹⁰:*

$$\begin{array}{c}
 \text{IMP} \frac{}{x(z).P \xrightarrow{x(y)} P\{y/z\}} \qquad \text{OUT} \frac{}{\bar{x}(y).P \xrightarrow{\bar{x}(y)} P} \qquad \text{TAU} \frac{}{\tau.P \xrightarrow{\tau} P} \\
 \\
 \text{OPEN} \frac{P \xrightarrow{\bar{x}(z)} P'}{(z)P \xrightarrow{\bar{x}[z]} P'} \quad z \neq x \qquad \text{RES} \frac{P \xrightarrow{\alpha} P'}{(z)P \xrightarrow{\alpha} (z)P'} \quad z \notin n(\alpha) \\
 \\
 \text{SUM-L} \frac{P \xrightarrow{\alpha} P'}{(P + Q) \xrightarrow{\alpha} P'} \qquad \text{PAR-L} \frac{P \xrightarrow{\alpha} P'}{(P \mid Q) \xrightarrow{\alpha} (P' \mid Q)} \quad bn(\alpha) \cap fn(Q) = \emptyset \\
 \\
 \text{COMM-L} \frac{P \xrightarrow{\bar{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{(P \mid Q) \xrightarrow{\tau} (P' \mid Q')} \qquad \text{CLOSE-L} \frac{P \xrightarrow{\bar{x}[z]} P' \quad Q \xrightarrow{x(z)} Q'}{(P \mid Q) \xrightarrow{\tau} (z)(P' \mid Q')} \quad z \notin fn(Q) \\
 \\
 \text{REP-ACT} \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P} \qquad \text{REP-COMM} \frac{P \xrightarrow{\bar{x}(y)} P' \quad P \xrightarrow{x(y)} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P} \\
 \\
 \text{REP-CLOSE} \frac{P \xrightarrow{\bar{x}[z]} P' \quad P \xrightarrow{x(z)} P''}{!P \xrightarrow{\tau} (z)(P' \mid P'') \mid !P} \quad z \notin fn(P)
 \end{array}$$

where $bn(P)$ is the set of names that are bound in P , $fn(P)$ is the set of names that have a free occurrence¹¹ in P and $n(P) = fn(P) \cup bn(P)$.

A system’s behaviour described by means of a π -calculus process P can be modelled as LTS (S, L, R) where S is a set of π -calculus processes derivable from P by means of the transition relations (each process represents a state of the LTS), L is a set of π -calculus actions according to Definition 12 (they represent labels of the LTS), and $R \subseteq S \times L \times S$ is a π -calculus transition relation between the processes according to Definition 13 (i.e. between the states by means of the labels of the LTS).

2.3.2 Congruences of Processes

In π -calculus, *congruences* are equivalence relations¹² on π -calculus processes, which allows to formulate structural and behavioural equivalences between

¹⁰ For the rules SUM-L, PAR-L, COMM-L, and CLOSE-L, there exist also their „right“ variants SUM-R, PAR-R, COMM-R, and CLOSE-R, respectively, where the (first) activity modifies process Q instead of process P .

¹¹ See also Definition 7.

¹² The *equivalences* are relations that are reflexive, symmetric, and transitive. The *congruences* ensure that if processes P and Q are in a relation of equivalence and process P is a subprocess (a component) of process R , then process R with

the processes. Two π -calculus processes express the same behaviour if they are *barbed congruent*, which means *bisimilar* in terms of labelled state transition systems, i.e. if no difference can be observed when they are put into an arbitrary π -calculus context and compared using the appropriate bisimulation game [SW03].

There are four important relations – namely an early strong bisimulation, a late strong bisimulation, an early weak bisimulation, and a late weak bisimulation. *Early and late bisimulations* differ in ways to treat input actions (see the early and late instantiation in Section 2.3.1). *Strong and weak bisimulations* differ in ways to treat internal actions, the strong bisimulation treats internal τ -action and visible action equally while the weak bisimulation makes abstraction from the number of internal τ -actions (i.e. evolution of bisimilar systems is independent on their internal τ -actions).

In this book, the input actions are treated as the early instantiation, therefore we formally define only the early bisimulations according to [MPW92, SW03].

Definition 14 (Strong bisimilarity/bisimulation/simulation) A relation \sim is defined as a strong (early) bisimilarity iff for $P \sim Q$ there exists a strong bisimulation \mathcal{S} such that PSQ . A binary relation \mathcal{S} is defined as a strong (early) bisimulation iff both \mathcal{S} and its inverse are strong simulations. The relation \mathcal{S} is defined as a strong (early) simulation iff PSQ implies that

1. if $P \xrightarrow{\alpha} P'$ and α is τ or $\bar{x}\langle y \rangle$ (i.e. a free action) where y is not a name in P or Q , then for some $Q', Q \xrightarrow{\alpha} Q'$ and $P'SQ'$,
2. if $P \xrightarrow{x(y)} P'$ and y is not a name in P or Q , then for all w , there is Q' such that $Q \xrightarrow{x(y)} Q'$ and $P' \{w/y\} SQ' \{w/y\}$,
3. if $P \xrightarrow{\bar{x}[y]} P'$ and y is not a name in P or Q , then for some $Q', Q \xrightarrow{\bar{x}[y]} Q'$ and $P'SQ'$.

Definition 15 (Weak bisimilarity/bisimulation/simulation) A relation \approx is defined as a weak (early) bisimilarity iff for $P \approx Q$ there exists a weak bisimulation \mathcal{S} such that PSQ . A binary relation \mathcal{S} is defined as a weak (early) bisimulation iff both \mathcal{S} and its inverse are weak simulations. The relation \mathcal{S} is defined as a weak (early) simulation iff PSQ implies that

1. if $P \xrightarrow{\alpha} P'$ and α is τ or $\bar{x}\langle y \rangle$ (i.e. a free action) where y is not a name in P or Q , then for some $Q', Q \xrightarrow{\tau^*} \xrightarrow{\alpha} \xrightarrow{\tau^*} Q'$ and $P'SQ'$,
2. if $P \xrightarrow{x(y)} P'$ and y is not a name in P or Q , then for all w , there is Q' such that $Q \xrightarrow{\tau^*} \xrightarrow{x(y)} \xrightarrow{\tau^*} Q'$ and $P' \{w/y\} SQ' \{w/y\}$,

substituted P for Q is in the relation of equivalence with the original process R (i.e. a substitution of equivalent components of processes does not break the equivalence of the processes).

3. if $P \xrightarrow{\bar{x}[y]} P'$ and y is not a name in P or Q , then for some Q' ,
- $$Q \xrightarrow{\tau^*} \bar{x}[y] \xrightarrow{\tau^*} Q' \text{ and } P'SQ',$$

where $\xrightarrow{\tau^*}$ is the reflexive and transitive closure¹³ of $\xrightarrow{\tau}$.

However, bisimilarity relations \sim and \approx are not congruence relations [SW03]. The reason is that all free names of related processes are open to instantiation, which is not handled by the bisimilarity relations. Therefore, an *open bisimilarity* is defined as a congruence relation as follows [BN07, SW03].

Definition 16 (Open D-bisimilarity and open bisimulation) Let P and Q be π -calculus processes, D is a distinction, and \mathcal{D} is a set of distinctions. We say that P and Q are open D-bisimilar, written $P \sim_o^D Q$, if there exists an open bisimulation $(S_D)_{D \in \mathcal{D}}$ such that $D \in \mathcal{D}$ and $(P, Q) \in S_D$. The family $(S_D)_{D \in \mathcal{D}}$ of symmetric relations is the open bisimulation if for all $D \in \mathcal{D}$, for all substitutions σ such that σ respects D , for all $(P, Q) \in S_D$, whenever $P\sigma \xrightarrow{\alpha} P'$ with names in $bn(\alpha)$ fresh, there exists Q' such that $Q\sigma \xrightarrow{\alpha} Q'$ and

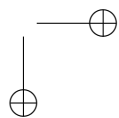
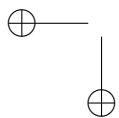
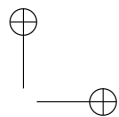
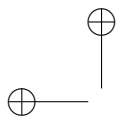
1. if $\alpha = \bar{a}[z]$ for some a and z , $D' \in \mathcal{D}$ and $(P', Q') \in S_{D'}$ where $D' = D\sigma \cup (\{z\} \times (fn((P + Q)\sigma) \cup n(D\sigma)))$,
2. otherwise, $D\sigma \in \mathcal{D}$ and $(P', Q') \in S_{D\sigma}$,

where $bn(\alpha)$ is the set of names that are bound in α , $fn(R)$ is the set of names that have a free occurrence¹⁴ in R , $n(R) = fn(R) \cup bn(R)$, σ respects D iff $x\sigma = y\sigma$ for all $(x, y) \in D$, and $D\sigma = \{(x\sigma, y\sigma) \mid (x, y) \in D\}$.

Open bisimilarity is useful for ascertaining automatically whether processes are bisimilar [SW03], i.e. open \emptyset -bisimilar (open D-bisimilar for $D = \emptyset$ according to Definition 16). This relation can be checked by means of several tools, which will be described in Section 8.2.

¹³ Informally, it means that there can be zero to many transitions $\xrightarrow{\tau}$ with internal τ -actions.

¹⁴ See also Definition 7.



3

Software Component Architecture

At the beginning of this chapter, we introduce software architectures in general in Section 3.1 and component-based development, which is a software development methodology strongly oriented on composability and reusability in software architecture, in Section 3.2. Finally, we analyse several important state-of-the-art works that deal with component-based development and components models in Section 3.3 and architecture description languages in Section 3.4. This chapter is particularly focused on the approaches that support features of dynamic and mobile architectures.

3.1 Software Architecture

The *software architecture* is defined by [IEE00] 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“. Other definition [BCK03] adds, that the architecture describes only externally visible properties of components, i.e. it is an abstraction of a system that suppresses details of components, except for services published by interfaces, relationships to environment of the components, and their externally observable behaviour.

The architecture of a software system can be described using several concurrent views [Kru95, IEE00] including particularly logical (structural) view and process (behavioural) view:

logical (structural) view describes logical structure of the system, e.g. an object model where an object oriented design method is used, or entity-relationship diagram where design of the system is data-driven;
process (behavioural) 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 [Oqu04]: static architecture, dynamic architecture, and mobile architecture. The last one is also known as a fully dynamic architecture.

static architecture – The architecture of a software system is static if there are no changes of the system’s structure during the system’s runtime. After initialisation of the system, there are no new connections between the system’s components and existing connections are not destroyed.

dynamic architecture – In the dynamic architecture, there exist rules of evolution of a software system in time (also called a „dynamics“). The system’s components and connections are created and destroyed during the system’s runtime according to the rules from the system’s design-time.

mobile architecture – 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“¹) according to rules from the system’s design-time and functional requirements.

3.2 Component-Based Development

The *component-based development*² (CBD, see [Szy02, CCL06]) 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* (CBS) 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*.

There is a difference between conception of „component“ from the CBD and „object“ from object-oriented programming [Szy02], although some common features exist (e.g. separation of interfaces from their implementations). An object is an instance of a class from a generalisation/specialisation hierarchy. It has a unique identity and an externally observable state (via object’s properties). A component is a self-contained entity (no classes or type hierarchy) without externally observable states. This, together with high context independence of components, increases re-usability beyond object oriented programming.

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

¹ The component mobility allows cloning and migration of the system’s components into different contexts.

² The CBD is also known as *component-based software engineering* (CBSE) or *component programming*.

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 in dynamic and mobile architectures includes [Oqu04]:

1. *actions*, which are consumed and produced by a system (inputs, outputs, and internal actions);
2. *relationships* between actions, how the input actions are processed by the system;
3. *changes* of an architecture according to the actions, i.e. processes of creation and destruction of 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*.

Although the CBD can be the right way to cope with problems of the distributed information systems, it has some limitations in formal description, which restrict the full support of mobile architectures. Those restrictions can be delimited by usage of formal bases that do not consider dynamic reconfigurations and component mobility, by strict isolation of control and business logic of components that does not allow full integration of dynamic reconfigurations into the components, etc.

3.3 Component Models

Component models are specific metamodels of software architectures supporting the CBD. The component models should define syntax, semantics, and composition of components [LW05]. 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 [LW06] 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. After a short description of a component model supporting static architectures, Wright in Section 3.3.1, we introduce contemporary component models supporting features of dynamic and mobile architectures, namely Darwin in Section 3.3.2, SOFA in Section 3.3.3 and its successor SOFA 2.0 in Section 3.3.4, and Fractal in Section 3.3.5.

3.3.1 Wright

Wright [AG96] is a component model, which uses the process calculus of *Communicating Sequential Processes* (CSP, see Section 2.2). The component model Wright defines a *component* with CSP semantics as a structure composed of two parts, an interface and a „component-spec“. The *interface* consists of a finite number of *ports*. Each port represents required input part or provided output part of the interface corresponding to a CSP process, together with an input or output event. The *component-spec* defines composition of interactions described by the ports and specifies the component’s function.

A *connector* is entity of Wright, which acts as a connection between a collection of components. It describes interaction of the components and consists of a finite set of roles and a glue specification. Each *role* is a CSP process, which describes expected behaviour of one component participating in the interaction (it refers to a port of such component). The *glue* composes processes of the roles into one CSP process, which describes how the participating components interact.

Finally, a *configuration* describes actual bindings of the components and the connectors. It consists of two parts, instances and attachments. The *instances* define actual (named) components and connectors, which participate in the configuration. The *attachments* bind ports of the participating components to roles of the participating connectors. Wright provides *hierarchical composition*, the whole configuration (at lower level) can be declared as a component (at higher level of the hierarchy).

The component model Wright provides *architectural styles*. An architectural style is an abstract component, which is described as a prototype configuration associating specific types of components with specific types of connectors. The architectural style can also define integrity constraints of participating entities and prescribe which ports of internal components have to be published as ports of the architectural style (i.e. ports of the abstract component).

As it has been mentioned above, the semantics of Wright entities is formally defined by means of CSP. The formal semantics of Wright defines a successfully terminating process \S and also permits input and output events with associated data as communication between Wright components. An input or output event e with data x can be written as a prefix operator $e?x \rightarrow$ or $e!x \rightarrow$, respectively, as it has been described in Section 2.2. The successfully terminating process \S is formally defined as a process that engages success event \surd and then stops (i.e. $\surd \rightarrow \text{STOP}$).

Limitations of Wright are given by the used formalism, e.g. CSP supports only systems with static architecture. However, Wright has introduced approaches to many interesting features of component-based systems such as distinction between components and connectors, definition of compatibility of a component with a connector (through interaction of processes of ports and connectors), and introduction of architectural styles.

3.3.2 Darwin and Tracta

The component model Darwin [MDEK95, Gia99] allows distributed systems to be hierarchically composed of sets of component instances and their interconnections at each level of the hierarchy.

In Darwin, a *component* is defined by means of its required and provided services (interfaces). The *services* provided and required by the component allow it to interact with other components. A *type* of the services can be specified, but Darwin does not interpret the service type information and is used only by the underlying distributed platform (an implementation). *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 respects context independence of the components—they can be specified, implemented, and tested independently without need of the rest of a system.

A *semantics* of Darwin language [MDEK95] was originally derived from a semantics of the process algebra π -calculus (see Section 2.3). A component is described as a parametric π -calculus process with the component’s services as parameters of the process. For connection of components, there are defined processes *Prov*, *Req*, and *Bind*:

$$Prov \stackrel{def}{=} (p, s)!.(p(x).\bar{x}(s)) \quad Req \stackrel{def}{=} (r, l).r(y).\bar{y}(l) \quad Bind \stackrel{def}{=} (r, p).\bar{r}(p)$$

Behavioural description of a connection between two components can be defined as follows. At first, a π -calculus processes describing behaviour of the first component is composed with process *Prov*(*p*, *s*) where parameter *s* represents the component’s provided service, while a process describing behaviour of the second component is composed with process *Req*(*r*, *l*) where parameter *l* represents the component’s required service. Finally, the resulting processes are composed together and with process *Bind*(*r*, *p*) describing the connection.

The semantics of Darwin allows to specify a subset of *dynamic architectures*. It permits dynamic instantiation of new components at runtime, but does not allow specification of dynamic bindings or component removal.

Progress of works on the semantics of Darwin has issued in the *Tracta approach* [Gia99]. The formal basis of Tracta are *Labelled Transition Systems* (LTSs, see Section 2.1) with the algebra of *Finite State Processes* (FSP). The FSP is a specification language with well-defined semantics in terms of LTSs. It is used for behavioural specification of especially primitive components as finite LTSs. Then, the LTS are hierarchically composed into behavioural description of composite components. For this purpose, Tracta introduces a *parallel composition operator*, „||“, which allows to compose two LTS processes in the undefined order (it is commutative and associative). Tracta also defines a *relabelling operator* for renaming of actions of LTSs and *operators interface and restriction* to reduce scope of visibility of the actions.

A component in Darwin, which is formally described by means of LTS in Tracta, can be checked against various properties. These properties may be expressed as Büchi automata or as LTL formulas (a linear temporal logic of actions, ALTL). Besides reachability analysis, Tracta also provides two analysis strategies for two types of the properties, *safety* and *liveness*.

The original semantics of Darwin using the π -calculus has formed basic features of the language. The Tracta approach maps the Darwin’s semantics into LTSs and FSP formalisms. Despite its support for only limited subset of dynamic architectures, Tracta provides an interesting component model, which has introduced usage of LTSs³.

3.3.3 SOFA

In the component model SOFA [PBJ98], a part of *SOFA project (Software Appliances)*, a software system is described as a *hierarchical composition* of primitive and composite components. A *component* is an instance of a *template*, which is described by its frame and architecture. The *frame* is a black-box specification view of the component defining its *provided* and *required* interfaces. Primitive components are directly implemented by a software system—they have a primitive architecture. The *architecture* of a composite component is a grey-box implementation view, which defines first level of nesting in the component. It describes direct subcomponents and their interconnections via interfaces.

The connections of the interfaces can be *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. Non-exempting connections can be realised via connectors, implicitly for simple connections or explicitly. Explicit *connectors* are described in a similar way as the components, by a frame and an architecture. The *connector frame* is a set of roles, i.e. interfaces, which are compatible with interfaces of components. The *connector architecture* can be simple (for primitive connectors), i.e. directly implemented by a software system, or compound (for composite connectors), which contains instances of other connectors and components.

SOFA uses a *Component Definition Language* (CDL, [Men98]), which extends features of OMG IDL [OMG98] to allow specification of software components. Behaviour of a component is formally described by means of *behaviour protocols* [Viš02]. Every communication (a method call) forms an *event*, e.g. event m for method m , which is denoted by one of *event tokens* according to its semantics: $!m\uparrow$, $?m\uparrow$, $!m\downarrow$, and $?m\downarrow$, for emitting and accepting a method call and emitting and accepting a return, respectively. A sequence

³ Afterwards, the LTSs has been used e.g. for formal description of the component model Fractal (see Section 3.3.5).

of event tokens forms a *trace* (e.g. $\langle !m\uparrow; ?m\downarrow \rangle$). A behaviour protocol is a regular-like expression on the set of all event tokens, generating the set of traces. Then, behaviour of a SOFA entity (its interface, frame, and architecture) can be described by a behaviour protocol, i.e. the set of all traces, which can be produced by the entity. The architecture protocols are generated automatically from architecture description by a *CDL compiler*.

Besides basic operators of regular expressions, *sequencing*, *alternative*, and *repetition*, a behaviour protocol may contain enhanced and composed operators. The enhanced operators are *and-parallel operator* for interleaving composition, *or-parallel operator* for sequential parallel composition, and *restriction operator* that omits restricted events from traces. The composed operators are *composition* and *adjustment*, which from two different kinds of interleaving parallel compositions with synchronisation via a given set of events.

SOFA defines a *protocol conformance relation* between an architecture protocol and a frame protocol. The relation expresses that the architecture protocol generates only traces that are allowed by the frame protocol. Faulty computation detection is another control mechanism of component composition, which introduces *error tokens* of three types: *bad activity*, *non activity*, and *divergence*. Those tokens describe errors in communication of components. Sets of possible error traces leading to the error tokens are generated during composing of two components via a *consent operator*.

Despite the fact that SOFA supports modelling of a static architecture, it allows *dynamic update* of a component during a system’s runtime. The update consists in change of implementation (i.e. an architecture) of the component by a new one. Compatibility of the implementations is guaranteed by the conformance relation of a protocol of the new architecture and the component’s frame protocol. During the update of a component, passivity of the component and atomicity of the update must be ensured. A designer can mark states of the component’s behaviour, which are safe for the update, by special *update tokens* in the component’s behaviour protocol.

3.3.4 SOFA 2.0

The SOFA 2.0 [BHP06] is a new version of component model SOFA (see Section 3.3.3), which 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.

Permitted dynamic reconfigurations are predefined at design-time by *reconfiguration patterns*. SOFA 2.0 allows three reconfiguration patterns [HP06]: nested factory, component removal, and utility interface. The *nested factory pattern* covers adding a new component and a new connection to an architecture. The new component can be created by a factory component as a result of a method invocation on this factory and becomes a sibling of a component

that initiated the creation. The *utility interface pattern* allows a component to define utility interfaces. The reference to an utility interface can be freely passed among components, and any component can establish a connection using this reference, independently of the component’s level in architecture hierarchy. Such feature brings into component-based development a feature of service-oriented architectures (SOAs, see [Erl05]) and SOA becomes a specific case of a component model where all components (services) are interconnected solely via their utility interfaces.

In SOFA 2.0, *control parts* of components are composed of microcomponents. The *microcomponents* [MB05] are minimal primitive components without controller parts. Interfaces required to establish bindings between the microcomponents, which are the only needed microcomponents’ control features, are implemented directly by content parts of the microcomponents. The microcomponent model allows to capture architecture of the controller parts of components, to express that a controller requires a certain control (micro)component (via required interface of a microcomponent), and to define exactly interconnections of control and functional parts of a component as connections between microcomponents of the control parts and components of the functional parts.

Finally, SOFA 2.0 introduces multiple communication styles [BHP06], which define functionality of connectors. There are *four communication styles*: remote method invocation, message passing, streaming, and distributed shared memory. From the knowledge of a connector’s communication style, only a specific type of binding can be permitted or intercomponent communication can be optimised by choosing an appropriate middleware. Therefore, SOFA 2.0 distinguishes *two classes of connectors*: design connectors and runtime connectors. The *design connectors* are described by communication styles and communication-related features associated with each component interface involved in the communication. The *runtime connectors* are the code artifacts used at runtime to implement the design connectors, which are created by a *connector generator* automatically from their design counterparts. The generation is performed at deployment-time, before preparing and launching an application, with complete knowledge of the application’s environment.

3.3.5 Fractal

The *component model Fractal* [BCS02, BCS04] is a general component composition framework with support of dynamic architectures. A Fractal *component* is formed out of two parts: a controller and a content. The *content* of a *composite component* is composed of a finite number of nested components. Those subcomponents are controlled by the *controller* (also called „a membrane“) of the enclosing component. The controller acts as a composition operator. A component with empty content is called a *primitive component*. A component can be *shared* as a subcomponent by several distinct components.

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. The operations can be *one-way operations* and *two-way operations* (i.e. without and with return of a result, respectively). The interfaces can be of two sorts: client and server (i.e. required and provided, respectively). Besides, 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.

The *attribute control* provides operations to get and set values of component’s attributes. The *binding control* serves for binding and unbinding the component’s external client interfaces to some server interfaces of other component. The *content control* provides operations to add and remove other components as the component’s subcomponents (on the places that are permitted by a controller). Finally, the *life-cycle control* provides operations to start and stop the component. Usage of the binding control and the content control is allowed only when the component is stopped.

The *binding* is a directed connection between components. A *primitive binding* is a connection between two components, the first component with a client interface and the second component with a server interface. The interfaces must be compatible—the type of the server interface must be a sub-type of the type of the client interface⁴. Combination of primitive bindings and an ordinary Fractal component can be used as a *composite binding*, i.e. as a connection (a connector) between several components. Binding between a client interface (*c*) and a server interface (*s*) can be of three types: *normal* (if *c* and *s* are external interfaces), *export* (internal interface *c* of a component is connected to external interface *s* of its subcomponent) and *import* (internal interface *s* of a component is connected to external interface *c* of its subcomponent).

Behaviour of Fractal components can be formally described by means of *parametrised networks of communicating automata* language [Bar05]. Behaviour of each primitive component is modelled as a finite state *parametrised labelled transition system* (PLTS). It is a LTS (see Definition 2 in Section 2.1) with *parametrised actions* as labels, a set of *global variables for a whole system*, and a set of *local variables for each state*. Besides a parametrised action, each label of PLTS contains also a guard (a boolean expression) of transitions with this label and a set of expressions, which assign values of variables of the transitions’ target states from free variables of the transitions’ source states and the global variables.

⁴ The server interface can accept at least all the operation invocations that the client interface can emit, and the client interface can accept (at least) all the returns from previously invoked operations on the server interface.

Behaviour of a composite component is defined using a *parametrised synchronisation network* (PNET), which acts as a generalised parallel operator of component composition. Arguments of such operator are *parametrised sorts*, which are sets of observable parametrised actions of subcomponents’ PLTSs. Besides those sorts, the pNet contains a set of global parametrised actions and a transducer.

The *transducer* is a PLTS, which is a synchronisation product (see Definition 5 in Section 2.1) of the subcomponents’ PLTSs. Each of its states corresponds to specific configuration of the subcomponents’ PLTSs, and each its transition is labelled by a synchronisation vector (see Definition 4 in Section 2.1) of actions of those PLTSs. During runtime of a composite component, when synchronised actions in a label of the transducer’s transition occurs, the transducer changes its state according to such transition. The change represents reconfiguration of the composite component’s architecture. The resulting behaviour of a composite component is computed as a product of subcomponents’ PLTSs and the transducer.

Behaviour of a Fractal component’s controller can be formally described by means of PLTS/PNET. The result is composition of PLTSs for binding and unbinding of each of the component’s functional interfaces (one PLTS per one interface) and PLTS for starting and stopping the component. The mentioned *formal approach requires* that the start/stop operations are recursive (they affect a component and each one of its subcomponents simultaneously), functional operations can not fire control operations, and a component’s external functional interfaces are simply forwarded to its internal interfaces (without any control capability).

As a proof of concept, [Bar05] introduces a formal description of *Fractive* [BCM03], which is a Fractal implementation using a *ProActive middleware* [BBC⁺06]. However, a mobile architecture, which is also supported by the ProActive, has not yet been addressed. For verification of resulting behaviour of a component-based system, there is introduced platform VER-CORS [BCMR06] integrating several tools. The fundamental tools are modified *Bandera Project* for generating PNETs from Java programs in ProActive and *FC2Instantiate* to get instances from the parametrised descriptions of PLTS/PNETs in FC2Parametrized format to output FC2 format, which can be translated into a native input format of several external verification tools based on process algebras.

3.4 Architecture Description Languages

Architecture description languages (ADLs, see [Ves93]) are languages for describing software systems’ architectures. They 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 (see Section 3.3), where they are used for description of a software system’s

architecture in terms of the component models⁵. Alternatively, ADLs can be realised without the component models, based directly on general principles of the component-based development (see Section 3.2).

In this section we aim at the ADLs that do not depend directly on component models. We introduce a general architecture interchange language ACME in Section 3.4.1, possible strategies for modelling of software architectures in UML in Section 3.4.2, and recent ArchWare ADL in Section 3.4.3.

3.4.1 ACME

A large number of ADLs have been proposed [MT00]: for modelling of software architectures within a particular domain, as general-purpose architecture modelling languages, with and without component models and formal bases, etc. Each one of the various set of ADLs defines its own capabilities of architecture specification, including specific definitions of basic characteristics and constructs of its architecture. Features of ADLs are delimited by particular domains, component models, and formal bases.

In order to unify architectural specifications across ADLs, an *architecture interchange language* ACME [GMW00] has been developed. It establishes a common basis for the ADLs and enables integration of their support tools. The ACME defines *core architectural entities*: components and connectors (as they are described in Section 3.2), 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).

3.4.2 Unified Modelling Language

Unified Modelling Language (UML, see [OMG05b, OMG07b]) can act as another approach to description of a software system’s architecture. The metamodeling architecture of UML suggests three possible strategies for modelling of software architectures [MRRR02]:

⁵ In some cases, the line between concepts of a component model and an ADL can be blurred (e.g. Wright, which is described in Section 3.3.1, can be also designated an ADL).

1. to use UML „as is“ – it results in architectural models that are immediately understandable by any UML user and manipulable by UML tools, but there are only limited methods explicitly representing the relationship between existing UML constructs and architectural concepts (such as connectors and architectural styles);
2. to constrain the UML metamodel using UML’s built-in extension mechanisms (constraints, stereotypes, profiles) – it explicitly represents and enforces architectural constraints, is manipulable by standard UML tools, and would be understandable to UML users, but exact specification of a modelling space can be difficult (i.e. extensions may not cover all features of the architectures);
3. to extend the UML metamodel to directly support the needed architectural concepts – it could fully capture every desired feature of every ADL and provide „native“ support of software architectures in UML (new modelling capabilities), but results in a notation that does not conform to the UML standard and could be incompatible with UML-compliant tools.

Each approach has certain potential advantages and disadvantages. Today, the second mentioned strategy for modelling of software architectures in UML is preferred, i.e. an ADL’s entities and their semantics are mapped into the UML 2 as the ADL’s UML profile. Resulting metamodels of component-based systems can be used to develop supporting tools, e.g. in *Eclipse Modeling Framework* (EMF) [BSM⁺03, Ecl07b] for modelling and code generation of tools based on component models, or in *Eclipse Graphical Modeling Framework* (GMF) [Ecl07a] for developing graphical editors according to the rules described in the component models’ metamodels (based on EMF).

UML 2 Components

The UML 2 has introduced description of hierarchical architecture of component-based systems [AGM04] by means of *structured classes*, i.e. the classes that allow nesting of other classes. In this section we review relevant concepts and UML 2 notation related to component modelling.

To separate specification of classes from their implementation, UML proposes *interfaces* [OMG05b], which describe operations of classes, their accessible attributes, possible associations under defined constraints and protocols. In UML 2 [OMG07b], the interfaces are drawn as specific classes stereotyped «**interface**». Classes that implement given interfaces are connected to them by „realisation“ relation (they provide the interfaces), while classes that access to the interfaces are connected to them by „dependency“ relation (they require the interfaces). In addition to „class“ style notation of stereotyped classes and their relations, UML provides „lollipop“ style notation. Both notations are shown in the example in Figure 3.1.

The components themselves are drawn as specific classes stereotyped «**component**» interconnected by means of „assembly connectors“ binding their

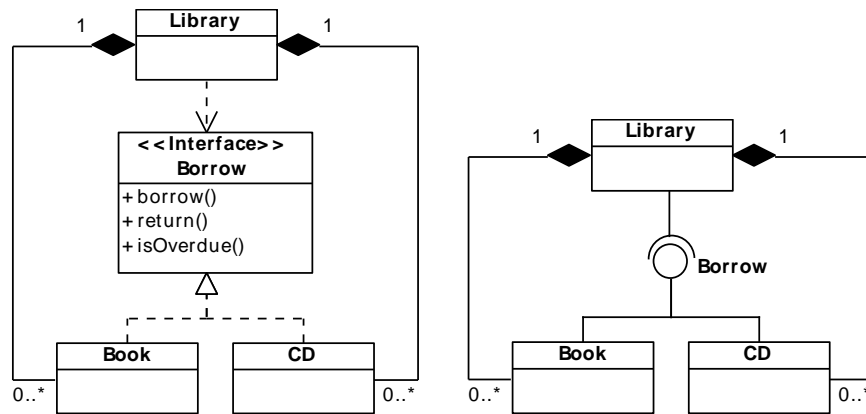


Fig. 3.1. An example of UML „class“ style notation with interface stereotypes and corresponding „lollipop“ style notation where **Borrow** acts as an „assembly connector“ between **Library** and **Book** or **CD** (the example is adopted from [AN05]).

interfaces in the „lollipop“ style notation. The UML 2 specification [OMG07b] states for such classes the following: „a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment“. The components can be used to represent many different entities, which are distinguished by stereotypes. For component-based development, the following stereotypes [AN05] can be important:

- **«specification»** – it specifies a domain of abstract components without defining their physical implementation;
- **«implementation»** – it specifies a separate implementation of a component, which has no specification itself, but has a dependency on a specific **«specification»** component;
- **«service»** – a stateless, functional component that computes a value;
- **«subsystem»** – a logical construct representing a unit of hierarchical decomposition that can not be instantiated at runtime.

As structured classes, UML 2 components can have internal structure and delegate their external interfaces to the parts of their internal assembly. External interfaces of a structured component are connected to its *ports* and by means of dependency relation to the corresponding internal interfaces (see an example in Figure 3.2).

It is highly recommended that architects use UML 2 components (and structured classes in general) to describe the hierarchical decomposition of component-based systems. However, the UML 2 does not explicitly provide all ADL’s constructs [Oqu05], e.g. for description of architectures with connectors or of dynamic architectures.

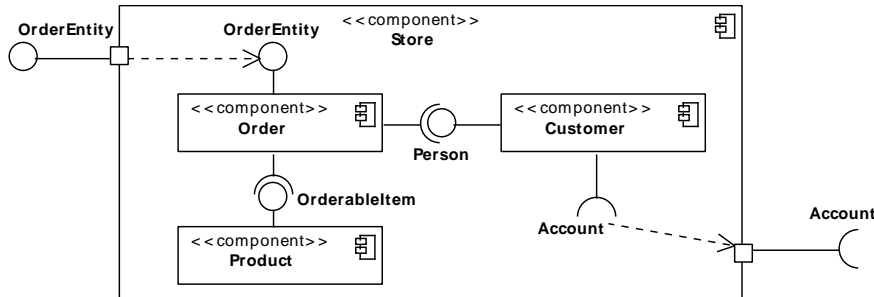


Fig. 3.2. An example of component *Store*, its internal structure and components *Order*, *Customer*, and *Product*, as parts of its internal assembly (the example is adopted from [OMG07b]).

3.4.3 ArchWare ADL

ArchWare [Arc06] was a 3 year project (since January 2002 to June 2005) funded by the European Community’s Fifth Framework Programme. The project was aimed to design, to develop, and to disseminate innovative architecture-centric languages, frameworks, and tools for engineering evolvable software systems. During the project, a formal architecture description language has been created, together with fitting analysis, refinement, and exchange languages. Those languages have been supported by appropriate frameworks and tools.

In this section, we describe the *ArchWare architecture description language* (ArchWare ADL, see [BMO⁺05]). The ArchWare ADL provides the 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:

1. π -ADL, which contains the core structure and behaviour constructs with the higher-order typed π -calculus as a formal basis (see Section 2.3);
2. $\sigma\pi$ -ADL, which contains style constructs for defining a base component-connector style and other derived styles, founded on top of the π -ADL;
3. $\mu\pi$ -AAL (Architecture Analysis Language), which is extension of the modal μ -calculus⁶ (with a predicate calculus) for description of behavioural and structural properties of communicating and mobile architectural elements.

The π -ADL [Oqu04] introduces a formal language for description of a dynamic software architecture’s elements. The π -ADL is formally defined by a formal transition and type system. The formal system is described

⁶ The μ -calculus allows to express properties of labelled transition systems by using the least and greatest fixed point operators.

in a layered approach: a formal system of a *base language* providing only behaviour constructs, a formal system of a *first-order language* extending the base language with value and structure constructs (base types and type constructors), and a formal system of a *higher-order language* extending the first-order language with the ability of the constructs to be declared, assigned, to have equality defined, and to be persistent.

The behaviour constructs from the base language copy the π -calculus constructs, which have been mentioned in Section 2.3. The base language contains „restrict“, „choose“, „compose“, „replicate“, and „unobservable“ constructs, „send“ and „receive“, „done“ behaviour (a null process), conditional behaviour, and a construct for renaming of names in behaviour. The base types are void, natural, integer, real, boolean, and string. The type constructors⁷ are tuple, view, union, any, quote, variant, location, and recursive, including iterable collection type constructors: sequence, set, and bag. Finally, the last behaviour construct is a connection of any type with support of mobility (also typical for the π -calculus). Moreover, the higher-order language defines behavioural abstraction and application, and behaviour definition (i.e. „a given name is defined as a given behaviour“).

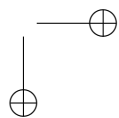
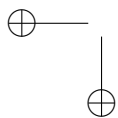
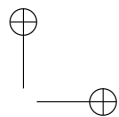
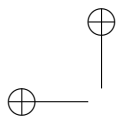
The $\sigma\pi$ -ADL [BMO⁺05] is realised as the outer layer of ArchWare ADL, which provides style constructs. It is formally constructed on a top of π -ADL and $\mu\pi$ -AAL, and it builds a bridge between those two languages. It allows definition of architectural element styles, represented by *property-guarded behaviour abstractions*, definition of domain specific extensions of the π -ADL or specific architectural patterns where their properties can be explicitly defined and preserved.

The $\sigma\pi$ -ADL is defined in two layers. The first (inner) layer is built on the top of π -ADL and introduces a partial application of behaviour abstractions, so-called *hierarchical abstractions*, allowing reuse of abstraction definitions at different levels of application. The second (outer) layer introduces possibility to declare properties, called *an architectural style* and expressed in $\mu\pi$ -AAL, as well as to attach the properties to behaviour abstractions. A property represents a constraint that is imposed to architectures that follow a specific architectural style (it defines a family of the architectures). The $\sigma\pi$ -ADL allows also to build *hierarchies of styles*.

During the ArchWare project, an *UML 2 profile* for ArchWare ADL has been developed [Oqu05], as well as several tools, e.g. for theorem proving and model checking⁸ of ArchWare ADL and for generating a code for ArchWare ADL models, which is executable by an *ArchWare virtual machine*. The ArchWare ADL has been applied in several case studies and also been used for designing and implementing the *ArchWare Software Engineering Environment*.

⁷ The view is a tuple with labelled elements, the quote is a label and the location is a named container for storing and retrieving values.

⁸ The tools are described in documents D3.5b and D3.6c in [Arc06].



4

Service Oriented Architecture

Service-oriented architecture (SOA, [Erl05]) is an architectural style for aligning business and IT architectures. It is a complex solution for analysis, design, maintaining, and integration of enterprise applications that are based on services.

This chapter deals with basic description of the service-oriented architecture. At the beginning, in Section 4.1, we introduce fundamental SOA principles and describe transformation of business processes in a Business Process Modeling Notation into UML service diagrams. Section 4.2 provides a short introduction to the services' implementation with focus on their communication. Finally, we describe a relation between services and component-based systems in Section 4.3. The chapter provides a basis for linking the services to underlying component-based systems, which will be proposed in the next part of this book.

4.1 Design of Services

Service-oriented architecture represents a model in which functionality is decomposed into small, distinct units, known as „services“, which can be distributed over a network and can be combined together and reused to create business applications [Erl05]. *Services* are defined as autonomous platform-independent entities enabling access to their capabilities via their provided interfaces. They can communicate:

1. **by passing data** between two services – in *service contracts*, services receiving data are *requesters*, while services sending the data are *providers*,
2. **by coordinating an activity** between two or more services – a multi-party collaboration between services is usually known as *service choreography* in case of the collaboration without a controlling service or as *service orchestration* if there exists a service that controls the collaboration.

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

Business processes describe the system as a hierarchically composed business process where each decomposable process (at each level of the composition) represents a sequence of steps in accordance with some business rules leading to a business aim¹.

Services implement business processes and their parts with well-defined interfaces and interoperability for the benefit of business. *Business (entity) services* encapsulate distinct sets of business logic, *utility services* provide generic, non-application specific, and reusable functionality, and *controller (task-centric) services* act as parent services to service composition members and ensure their assembly and coordination to execution of an overall business task [Erl05].

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

4.1.1 Business Process Modelling

Communication of services in SOA is aimed for the benefit of business. A new designed service has to meet business requirements that are traditionally specified by a business process model represented as a *business process diagram* (BPD). The diagram should capture which business processes are going to be done, who is going to do them, when and where will they be done, how and why will they be done, and who is dependent on their being done [CKO92]. A *business process* is a sequence of structured activities (actions) leading to a specific business aim. The *activities* have their own attributes and can be decomposed into several collaborating sub-processes at a lower level of abstraction.

There are several notations [LK06] for describing business process models and drawing business process diagrams (e.g. Business Process Definition Meta-model [OMG08a], Event Driven Process Chain [Sch00], IDEF3 [MMP⁺95], Petri Net [SW01], and UML 2 Activity Diagram [OMG07b]). However, *Business Process Modelling Notation* (BPMN, [OMG08b]) has played the most dominant role in the past several years. It is a standard, readily understandable notation, which allows transformation into an execution language, namely the *Business Process Execution Language for Web Services* (BPEL4WS, [ACD⁺03, ODvdAtH06]), an application of BPEL.

Business process modelling depends on a specific notation of business process models. It includes decomposing of business processes into their most detailed representations, resulting in series of granular actions. Actions that

¹ Business requirements are traditionally specified by a *business process model* (BPM).

are suitable for service encapsulation become potential service capability candidates [Erl05].

4.1.2 Business-to-Service Transformation

According to [Ars04], the initial activity in development of a new SOA-based system is a *service identification* [IB07], which is a part of service-oriented analysis. It consists of a combination of top-down, bottom-up, and middle-out techniques of domain decomposition of legacy systems, asset analysis, and goal-service modelling when service capability candidates are grouped into services. The result of the service identification is a set of candidate services (*business services* that encapsulate distinct sets of business logic, see the three levels of abstraction in Section 4.1).

In the context of service oriented design, the service identification is a prerequisite for the *business-to-service transformation*. Initially, a business process (BP), which is represented by a business process diagram (BPD) as an input of the transformation, is decomposed into individual tasks. Then, the transformation consists of two steps:

1. The first step is to identify which tasks from the BPD represent service invocations and therefore will be modeled as services in service diagrams. This decision is closely related to the service identification and takes into account such aspects as possible runtime scenarios, functionality of service providers, quality of service requirements, security issues, etc. [Ars04]
2. The transformation process itself [RW08] is based on a technique, which is introduced in [Ams05]. The technique integrates business process modelling and object modelling by providing a *business services model* (BSM) that is a mediator between business requirements and their implementation.

In the second step and according to [RW08], each service is modeled as an UML 2 component (see Section 3.4.2) with additional stereotype «*service*», which interacts with its environment via *interfaces* with stereotypes «*interface*». During the interaction, the service can act in two different roles: as a *service provider* or as a *service consumer*. These two roles are distinguished in the service model by means of different ports. *Provider ports* of a service implement interfaces that specify functional capabilities provided to possible consumers of the service, while *consumer ports* require interfaces of specific services to consume their functionality. Relationships between services and interfaces are stereotyped as «*use*» for interfaces of required services and by UML implementation relations for provided interfaces.

Finally, in addition to *business services*, which have been derived from predefined business entities in the previous step, *utility services* and *controller services* are created and modelled in UML in a similar way. Controller services are designed to controll service contracts and to finalize the required composition logic [Erl05] (see the services level of abstraction in Section 4.1).

4.1.3 Service Composition

Design of controller services puts less emphasis on exploring reusability, while it is more focused on services' roles as parent controllers [Erl05]. These services orchestrate subordinate services. However, from the structural point of view, SOA is a *flat model* where „composite“ services do not enclose their „internal“ services participating in the orchestration.

To fulfil the flat model and to support reusability of controller services in general, it is useful to design them as „stateless services“. Nevertheless, in the case of controller services, there are some problems we must cope with. The problems are related to the ability of controller services to synchronise actions of their subordinate services during their orchestration, e.g. to hold data between individual calls of individual subordinate services that are participating in the orchestration, realisation of a controller service with multiple interfaces that will be invoked in a specific order, etc. The solution can be to encode and to store a (hidden) state of a control service into values of parameters of a subordinate service's invocation, which will be required later, as a return value of the invocation, and utilised during next processing when the state of the control service will be restored [RW08].

Both, the flat model of SOA and statelessness of its services, put more importance on adequate design of the service composition as well as design of individual services.

4.2 Implementation of Services

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

- **remote procedure calls (RPC)** where the emphasis is on services' interfaces with strictly defined properties determining their compatibility (e.g. SOAP [GHM⁺07], JSON-RPC [AK06], and XML-RPC [Win99]);
- **resource oriented services** where predefined interfaces are independent on actual types of transferred resources, objects represented by unique identifiers, so that each of them is interacted with in the same way (e.g. Representational State Transfer, REST [Fie00]);
- **syndication-style publishing** where interfaces respect given standards for capturing all messages (e.g. Atom Publishing Protocol [Gdh07] and RSS [RSS07]);
- **vendor-specific services** where generic RPC capabilities are difficult to use (e.g. Oracle Database SOAP [PGG⁺06]).

Probably the most widely used implementation of SOA are *Web Services*. They are built on top of XML as a language for the data exchange and of SOAP as a framework for exchanging information. Individual Web services are described by means of *Web Services Description Language* (WSDL,

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

Table 4.1. The comparison of Service Oriented Architecture (SOA) and Component-Based Development and Systems (CBD/CBS), which has been published in [Ryc08].

[CMRW07]), which provides a component model and defines XML format of services. Operations and messages of Web services are described abstractly and then bound to concrete network protocols SOAP and HTTP and to message format MIME to define specific endpoints. The specific endpoints are combined into abstract endpoints, which form individual services.

Finally, *service brokers (service registries)* store information about available service providers for potential service requesters. Web Services uses *Universal Description, Discovery and Integration (UDDI, [CHvRR04])* registries of WSDL documents, which describe specific Web Services.

4.3 Services and Components

While the design of services in SOA is business oriented, components in Component-Based Development (CBD, see Section 3.2) are implementation oriented and usually need not respect any business rules or aims. Component-based systems are defined only by their initial configuration, component hierarchy (encapsulation), and components’ behaviour.

Table 4.1 compares features of SOA and Component-Based Development and Systems (CBD/CBS) from an implementation point of view—in aspects of communication of entities, description of their interconnections (i.e. their architecture), composition of entities, compatibility of their interfaces, and visibility of their states (i.e. „statefulness“ or „statelessness“ of the entities). For a detailed comparison, see [CCC⁺07].

4.3.1 Service Component Architecture

Service Component Architecture (SCA, [OSO07b, OSO07c, OSO07a]) is a general approach for design and implementation of SOA as component-based systems. It is a set of specifications [Ope08], which define a common mechanism for assembling of components and services into SOA applications using a wide range of technologies.

The SCA provides a model for *assembling of service components* and a model for *creation of component-based services*. In accordance with these models, a process of architectural design and implementation of a SOA application can be divided into two steps [Cha07]:

1. service components are implemented in such a way that each of them provides specific services and consumes other services;
2. sets of components are combined at design-time into composites, which are interconnected by wiring of service references to services.

In the first step, services of SOA are implemented in SCA as *service components* realising some business logic of a business application. The *components* [OSO07a, Cha07] offer capabilities through *service* interfaces and consume functions offered by other components through *reference* interfaces. Each interface offers or refers a number of *operations* and is described by means of a specific technology (e.g. in WSDL for a component implemented in BPEL, see Section 4.2 and Section 4.1.1). Moreover, a component can have one or more *properties* with values specific for individual instantiations of the component.

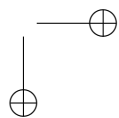
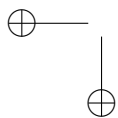
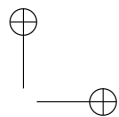
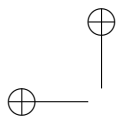
In the second step, the SCA components are combined at design-time into larger logical structures called *composites* [OSO07a, Cha07, OSO07b]. They are logical constructs for design purposes and usually do not determine the components' distribution at a runtime. A composite can be described in a XML-based *Service Component Definition Language* (SCDL, [OSO07a, Ope07]) as a set of its services, references, and properties, included internal components with their services, references, and properties, and a set of wires. The *wires* connect source component references to target component services in a case of connections of two components or promote composite references to internal component references and composite services to internal component service in a case of composites and their components.

Finally, service components are grouped into SCA *domains*, which represent complete runtime configurations potentially distributed over a series of interconnected runtime nodes [OSO07a]. Components in the same domain must be implemented by the same technology and can be interconnected directly by means of SCA wires. In a case of connections of services inside and outside a domain, there must be bound specific mechanisms for addressing and accessing the services (e.g. WSDL endpoint URIs in a case of Web services technology). External clients of a service that is developed and deployed using SCA should not be able to tell that SCA was used to implement the service—the SCA is an implementation detail [OSO07a].

4.3 Services and Components 43

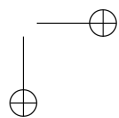
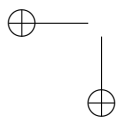
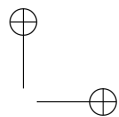
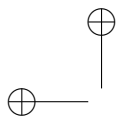
The SCA [Ope08] allows to describe service components in programming languages Java, C, C++, COBOL, and WS-BPEL. It supports environments and frameworks Spring and Java EE. Services can be accessed as Web services, Java Message Services, Enterprise JavaBeans, and J2EE Connector Architecture entities. Reference implementations are provided in „traditional“ programming languages such as Java, C++, and BPEL, but also in scripting languages such as PHP and JavaScript and in declarative languages such as XQuery and SQL [OSO07a].

The SCA allows to implement and assemble services at a business level and suppress implementation details of infrastructure capabilities and access methods used to invoke the services. However, in comparison with the component models from Section 3.3, the SCA does not provide any formalism for description of behaviour of component services and their composites.



Part II

Component Model for Mobile Architectures



5

Component Model

The previous part of this book has dealt with the state-of-the-art review of software component architecture in Chapter 3 and service-oriented architecture in Chapter 4. In this part, we will build on previous works and introduce a high-level component model for mobile architectures addressing the current issues of the existing component models and architecture description languages (see Section 3.3 and Section 3.4, respectively). The issues are related to the problem factors F1–F5, which have been described in Section 1.1. The component model allows dynamic reconfiguration, component mobility, and a specific combination of control and business logic of components. Behavioural description of individual components and their mutual communication is based on the calculus of mobile processes from Section 2.3.

The component model can be presented in two views¹: logical (structural) view and process (behavioural) view. At first, in Section 5.1, we introduce the component model’s metamodel, which describes basic entities of the component model and their relations and features. The second view, in Section 5.2, is focused on behaviour of the component model’s entities, especially on component mobility. Finally, in Section 5.3, an example of a component-based system is introduced and its structure and behaviour are described.

5.1 Logical View

The component model for mobile architectures is described as a metamodel in the context of a four-layer modelling architecture [OMG05a]. The metamodel is implemented in OMG’s *Meta Object Facility* (MOF, [OMG06a]), 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

¹ The scope of the views is described in Section 3.1.

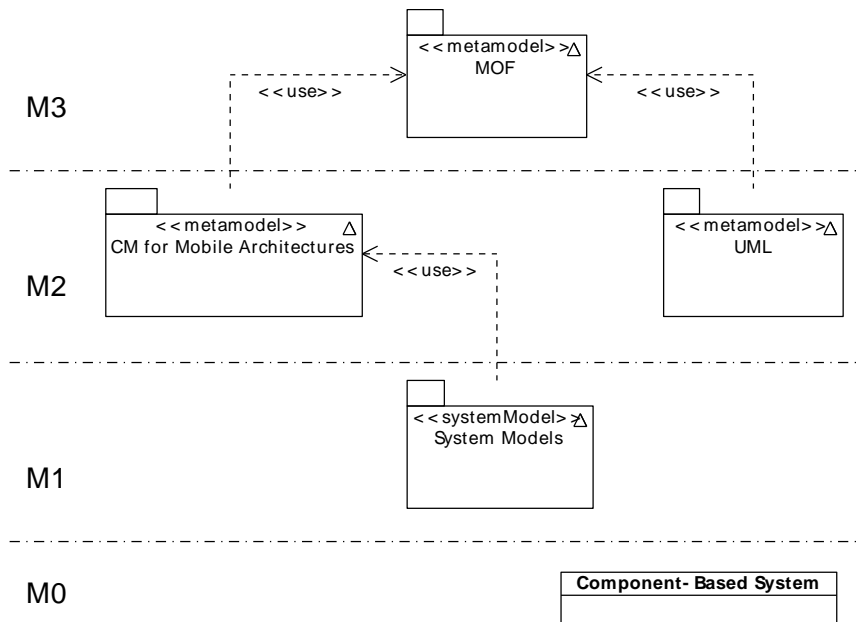


Fig. 5.1. The four-layer modelling architecture of the component model and UML as metamodels in layer M2 and MOF as a meta-metamodel in layer M3 (UML 2 notation).

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.

The four-layer modelling architecture is shown in Figure 5.1. 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».

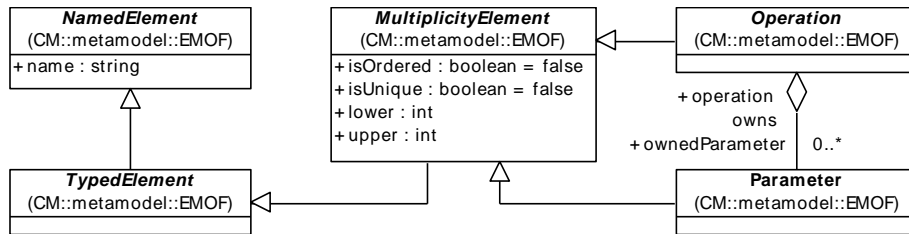


Fig. 5.2. A simplified part of the EMOF metamodel [OMG06a] with classes that will be extended by the component model.

5.1.1 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, [OMG06a]). MOF is in layer M3 in the four-layer modelling architecture (see Section 5.1). 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 [OMG07a]). For purposes of this chapter, 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`, `EMOF::TypedElement`, and `EMOF::Operation`, which are outlined in Figure 5.2. A complete and detailed definition of the EMOF classes can be found in [OMG06a].

Components and Interfaces

Figure 5.3 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.

50 5 Component Model

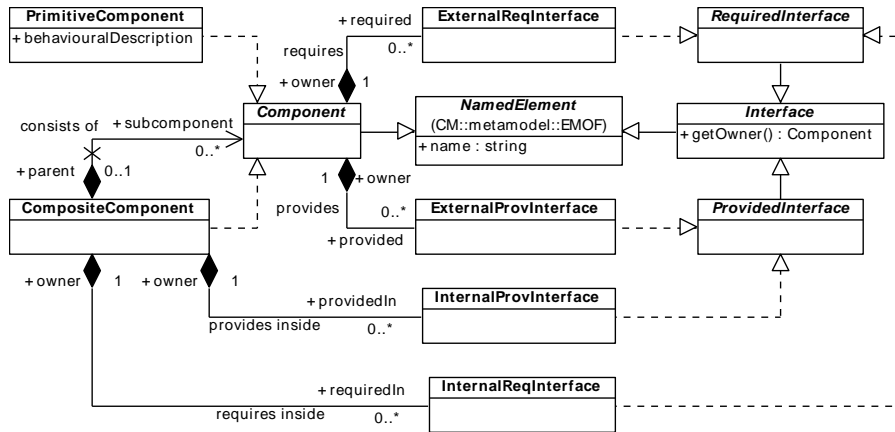


Fig. 5.3. Abstract component, realisations, and interfaces, extending `EMOF::NamedElement` in the metamodel of the component model.

In our approach, a *component*, which is an active communicating entity of a component-based software system, can be described from 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).

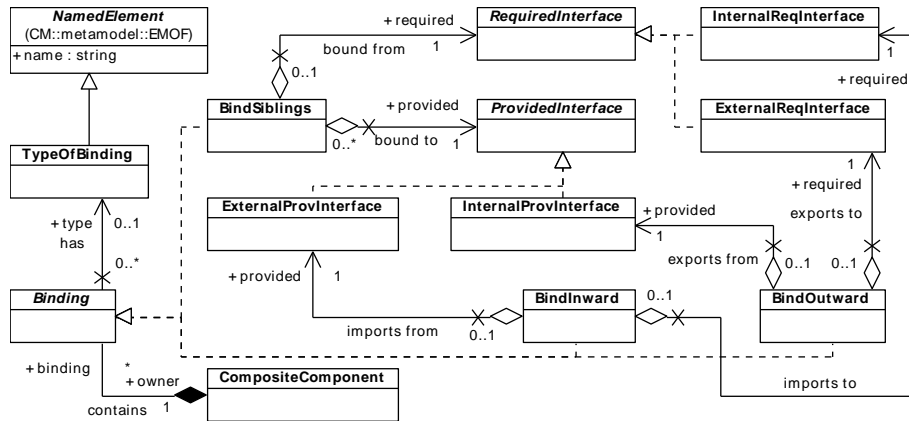


Fig. 5.4. Binding and its different realisations between interfaces of a composite component realisation in the metamodel of the component model. Classes `CompositeComponent` and `...Interface` are identical to the classes in Figure 5.3.

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 5.4. 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 realisation². The binding interconnects required interfaces (class `RequiredInterface`) via relations „bound from“ to provided interfaces (class `ProvidedInterfaces`) via relations „bound to“.
2. Binding of external provided interfaces of a composite component realisation to its internal required interfaces is represented by class `BindInward`. The external interfaces are provided to neighbouring components of the composite component acting as an abstract component (relation „imports from“ an instance of class `ExternalProvInterface`), while the internal

² The diagram in Figure 5.4 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.

52 5 Component Model

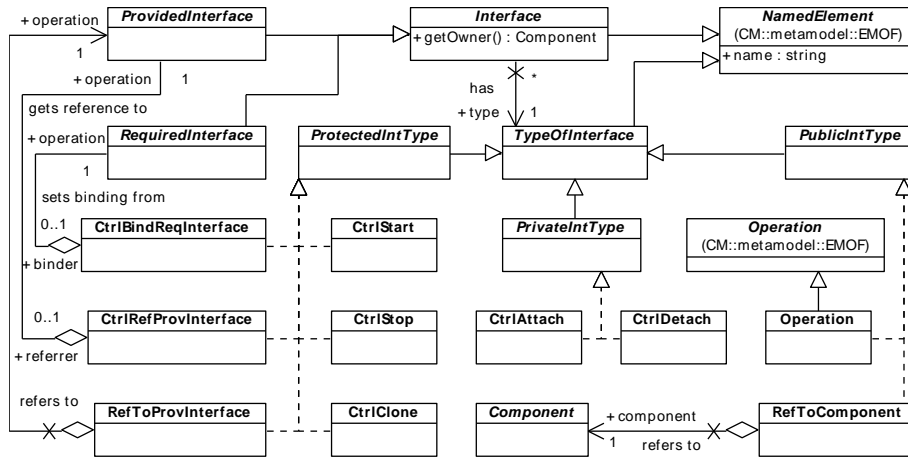


Fig. 5.5. Types of interfaces with class `Operation` extending `EMOF::Operation` in the metamodel of the component model. Classes `Interface`, `ProvidedInterface`, `RequiredInterface`, and `Component` are identical to the classes in Figure 5.3.

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 5.5.

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 interfaces³. 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 (see Figure 5.5):

- 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.
- Protected interface `CtrlRefProvInterface` provides references to given provided interface `ProvidedInterface` of type `Operation`⁴, while protected interface `CtrlBindReqInterface` allows to establish a new binding of specific required interface `RequiredInterface` of type `Operation`⁴ 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`⁴, while public interface

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

`RefToComponent` allows to pass references of a whole component `Component`, which is required to support component mobility.

Interfaces of type `Operation` are also known as *functional interfaces*, while the others are known as *control interfaces*⁵.

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 5.4. 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 5.5. 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 5.1.1).

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

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

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.

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

⁵ The functional and control interfaces in the metamodel can be compared with the functional and control interfaces in Section 3.3.5.

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

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

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

$$(\forall c : \text{CompositeComponent}) (\\ ((\forall b : \text{BindInward} \in c.\text{binding}) \\ (b.\text{provided.type} \text{ is } \text{PublicIntType} \\ \wedge b.\text{required.type} \text{ is } \text{PublicIntType}) \\) \wedge ((\forall b : \text{BindOutward} \in c.\text{binding}) \\ (b.\text{provided.type} \text{ is } \text{PublicIntType} \\ \wedge b.\text{required.type} \text{ is } \text{PublicIntType}) \\) \\)$$

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.

$$(\forall c : \text{CompositeComponent}) (\forall b : \text{BindSiblings} \in c.\text{binding}) \\ (b.\text{provided.type} \text{ is } \text{PrivateIntType} \Rightarrow b.\text{provided} \in c.\text{providedIn})$$

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.

$$(\forall t : \text{CtrlBindReqInterface}) (t.\text{operation.type} \text{ is } \text{Operation}) \\ \wedge (\forall t : \text{CtrlRefProvInterface}) (t.\text{operation.type} \text{ is } \text{Operation}) \\ \wedge (\forall t : \text{RefToProvInterface}) (t.\text{operation.type} \text{ is } \text{Operation})$$

5.1.2 System Model

The component model’s metamodel is a model of layer M2 where it exists alongside UML (see Figure 5.1 in Section 5.1). 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 metamodel⁶.

A system model, as a model of layer M1 in the four-layer modelling architecture, can be described in UML in two contexts:

⁶ The aim of the utilisation is to reuse the well-established UML notation, although it is not formally defined as an UML profile (see Section 3.4.2).

56 5 Component Model

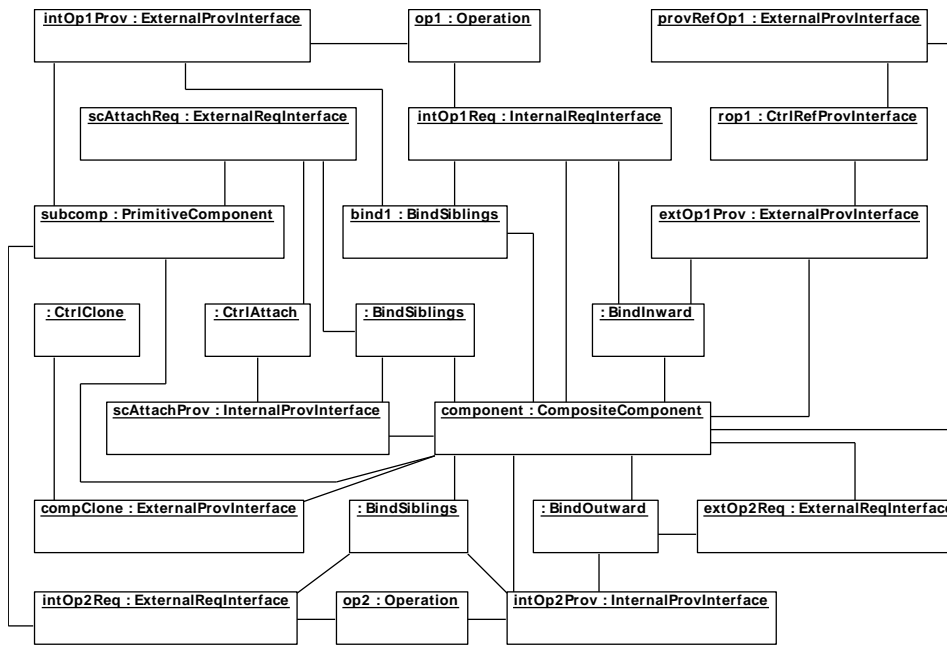


Fig. 5.6. The example of description of a system model as an object diagram with instances of classes from the component model’s metamodel.

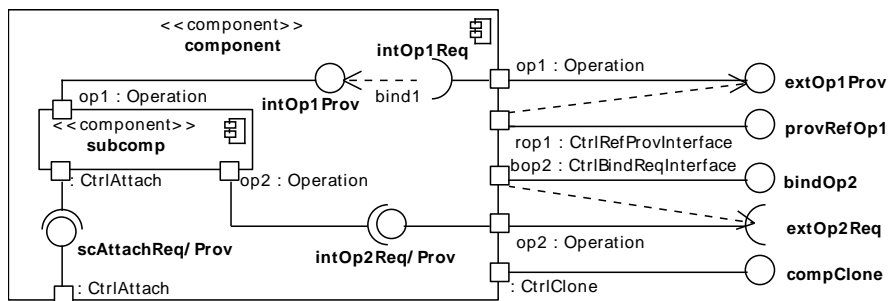


Fig. 5.7. An example of proposed notation of a system model in layer M1 by means of the component model from layer M0.

1. as an object diagram where objects in M1 layer are instances of classes from a metamodel in layer M2 (see Figure 5.6);
2. as a specific class diagram where entities in layer M0 are instances of classes from a diagram in layer M1 (see Figure 5.7).

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 5.6. 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 5.7. 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 5.7).
- **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 5.7).
- **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 5.7).
- **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 5.7).
- **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 5.7 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 5.7). 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 5.7).
- **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 5.7 provides references to interface `extOp1Prov` of the same component; analogously for the port of interface `bindOp2`).

5.2 Process View

In this section, the component model is presented in the process view (see Section 3.1). Behaviour of individual components and their mutual communication is described by means of the π -calculus (see Section 2.3).

According to the metamodel from Section 5.1.1, 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 5.3 in Section 5.1.1). 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 sub-

components (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.).

5.2.1 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: $s_0, s_1, c, p_1^s, \dots, p_n^s, p_1^g, \dots, p_m^g$;
- the internal view of a composite component only: $a, p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g$.

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 stopping⁷ (it represents an interface of type `CtrlStop` in the metamodel),
- via s_1 – a stopped component accepts a request for its starting⁷ (it represents 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 (it represents 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 (it represents 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 (it represents 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).

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

We should remark that there is 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, \dots, r_n (required) and p_1, \dots, p_m (provided) accessible via names p_1^s, \dots, p_n^s and p_1^g, \dots, p_m^g , respectively, to internal functional interfaces p'_1, \dots, p'_n (provided) and r'_1, \dots, r'_m (required) accessible via names p_1^g, \dots, p_n^g and p_1^s, \dots, 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 5.4 in Section 5.1.1).

5.2.2 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).

$$Wire \triangleq (x, y, d).(x(m).\bar{y}\langle m \rangle.Wire[x, y, d] + d)$$

Binding of components' functional interfaces is done via their control interfaces. These control interfaces allow to get a reference to a component's functional provided interface (via an interface of type `CtrlRefProvInterface` in the metamodel) and use the reference to bind a functional required interface of another component (via an interface of type `CtrlBindReqInterface` in the metamodel). Process *CtrlIfs* describes processing of requests via the control interfaces as follows:

$$\begin{aligned} SetIf &\triangleq (r, s, d).s(p).(\bar{d}.Wire[r, p, d] \mid SetIf[r, s, d]) \\ GetIf &\stackrel{def}{=} (p, g).g(r).\bar{r}\langle p \rangle \\ Plug &\stackrel{def}{=} (d).d \\ CtrlIfs &\stackrel{def}{=} (r_1, \dots, r_n, p_1^s, \dots, p_n^s, p_1, \dots, p_m, p_1^g, \dots, p_m^g). \\ &\quad \left(\prod_{i=1}^n (r_i^d)(Plug\langle r_i^d \rangle \mid SetIf[r_i, p_i^s, r_i^d]) \mid \prod_{j=1}^m !GetIf\langle p_j, p_j^g \rangle \right) \end{aligned}$$

where names $r_1, \dots, r_n, p_1^s, \dots, p_n^s, p_1, \dots, p_m, p_1^g, \dots, p_m^g$ have been defined in Section 5.2.1. Let us assume *CtrlIfs* shares its names r_1, \dots, r_n and p_1, \dots, p_m with a process describing a component's core functionality via its required and

⁸ The process will be used also in the following parts of Section 5.2.

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 $SetIf[r_i, p_i^s, r_i^d]$ 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 $r_1, \dots, r_n, p_1, \dots, p_m$ are connected to the names representing internal functional interfaces $p'_1, \dots, p'_n, r'_1, \dots, r'_m$. 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 $Ctrl_{EI}$:

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

5.2.3 Control of a Component's Life-cycle

Control of a composite component's life-cycle⁹ can be described as process $Ctrl_{SS}$.

$$Dist \stackrel{\Delta}{=} (p, m, r).(\bar{p}\langle m \rangle. Dist[p, m, r] + \bar{r}) \\ Life \stackrel{\Delta}{=} (s_x, s_y, p_x, p_y).s_x(m).(r)(Dist[p_x, m, r] \mid r.Life[s_y, s_x, p_y, p_x]) \\ Attach \stackrel{def}{=} (a, p_0, p_1).a(c_0, c_1, c_d)(d) \\ (c_d(m).\bar{d}\langle m \rangle.\bar{d}\langle m \rangle \mid Wire[p_0, c_0, d] \mid Wire[p_1, c_1, d]) \\ Ctrl_{SS} \stackrel{def}{=} (s_0, s_1, a).(p_0, p_1)(Life[s_1, s_0, p_1, p_0] \mid !Attach\langle a, p_0, p_1 \rangle)$$

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 **CtrlStart** 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 $Life[s_1, s_0, p_1, p_0]$ 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 $Dist[p_x, m, r]$ via shared name p_x , which can be p_0 in a case the component

⁹ A primitive component handles stop and start interfaces directly.

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.

5.2.4 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 another component. Process $Ctrl_{clone}$ describes processing of requests for cloning of a component as follows:

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

where pseudo-application $Component\langle s_0, s_1, c, p_1^s, \dots, p_n^s, p_1^g, \dots, p_m^g \rangle$ 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, \dots, p_n^s via name r and names p_1^g, \dots, 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 update¹⁰, 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 $Update$:

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

$$\begin{aligned}
 Update \triangleq & (u, a, s_0, s_d, p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g).(k, s'_d) \\
 & (\bar{u}\langle k \rangle.k(s'_0, s'_1, c, r', p').\bar{s}_0.\bar{a}\langle s'_0, s'_1, s'_d \rangle.\bar{s}_d \\
 & .r'(p_1^s, \dots, p_n^s).(x)(\bar{p}_1^g\langle x \rangle.x(p).\bar{p}_1^s\langle p \rangle \dots \bar{p}_n^g\langle x \rangle.x(p).\bar{p}_n^s\langle p \rangle) \\
 & .p'(p_1^g, \dots, p_m^g).(x)(\bar{p}_1^g\langle x \rangle.x(p).\bar{p}_1^s\langle p \rangle \dots \bar{p}_n^g\langle x \rangle.x(p).\bar{p}_m^s\langle p \rangle) \\
 & .\bar{s}_1.Update[u, a, s'_0, s'_d, p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g])
 \end{aligned}$$

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 $p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g$ represent a context of the old subcomponent, i.e. interfaces of neighbouring subcomponents, which have to be rebound to interfaces of the new component.

5.2.5 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 $s_0, s_1, r_1, \dots, r_n, p_1, \dots, p_m$ describes behaviour of the core of a primitive component (i.e. excluding behaviour of processing of its control actions), as it is defined by the component's developer. Further, let us assume that process abstraction $Comp_{subcomps}$ with parameters $a, p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g$ describes behaviour of a system of subcomponents interconnected by means of their interfaces into a composite component (see Section 5.2.2). Names $s_0, s_1, r_1, \dots, r_n, p_1, \dots, p_m$ and names $a, p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g$ are defined in Section 5.2.1.

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

$$\begin{aligned}
 Comp_{prim} \stackrel{def}{=} & (s_0, s_1, c, p_1^s, \dots, p_n^s, p_1^g, \dots, p_m^g).(r_1, \dots, r_n, p_1, \dots, p_m) \\
 & (Ctrl_{Ifs}\langle r_1, \dots, r_n, p_1^s, \dots, p_n^s, p_1, \dots, p_m, p_1^g, \dots, p_m^g \rangle \\
 & \quad | Ctrl_{clone}[c] \\
 & \quad | Comp_{impl}\langle s_0, s_1, r_1, \dots, r_n, p_1, \dots, p_m \rangle)
 \end{aligned}$$

64 5 Component Model

$$\begin{aligned}
 Comp_{comp} \stackrel{def}{=} & (s_0, s_1, c, p_1^s, \dots, p_n^s, p_1^g, \dots, p_m^g). \\
 & (a, r_1, \dots, r_n, p_1, \dots, p_m, r'_1, \dots, r'_m, p'_1, \dots, p'_n) \\
 & (Ctrl_{Ifs}\langle r_1, \dots, r_n, p_1^s, \dots, p_n^s, p_1, \dots, p_m, p_1^g, \dots, p_m^g \rangle \\
 & | Ctrl_{Ifs}\langle r'_1, \dots, r'_m, p_1^s, \dots, p_m^s, p'_1, \dots, p'_n, p_1^g, \dots, p_n^g \rangle \\
 & | Ctrl_{EI}\langle r_1, \dots, r_n, p_1, \dots, p_m, r'_1, \dots, r'_m, p'_1, \dots, p'_n \rangle \\
 & | Ctrl_{clone}[c] \\
 & | Ctrl_{SS}\langle s_0, s_1, a \rangle | Comp_{subcomps}\langle a, p_1^s, \dots, p_m^s, p_1^g, \dots, p_n^g \rangle)
 \end{aligned}$$

where the pseudo-applications of $Ctrl_{Ifs}$ represent behaviour of control parts of the components related to their functional interfaces (see Section 5.2.2), the constant applications of $Ctrl_{clone}$ describe behaviour of control parts of the components related to their cloning (see Section 5.2.4), the pseudo-application of $Ctrl_{SS}$ represents behaviour of the composite component’s control part processing its stop and start requests (see Section 5.2.3), and the pseudo-application of $Ctrl_{EI}$ describes communication between internal and external functional interfaces of the composite component (see Section 5.2.2).

5.3 An Example of a Component-Based System and its Description

As an example, we will describe a simple component-based system that dynamically changes its behaviour. At first, the system receives an input from an user as pair $(username, password)$ and verifies the user’s password in order to check the user’s identity. If the user’s credentials passes the verification, the system creates its fresh subcomponent providing user-specific functionality by initialising a clone of a user-specific component in a predefined state. The functionality of the new subcomponent is offered via the system’s external interface, i.e. the system offers different functionality via its external interface according to the user’s credentials received during the initialisation (as it is described above). The functionality can change after next initialisation with a different user’s credentials. Between initialisations, the subcomponent’s state can vary and affect behaviour of the whole system.

The component-based system is represented by component system composed of

- component *init* that verifies a user’s credentials and initiating user-specific functionality,
- component *workerA* that provides functionality specific for user „A“,
- component *workerB* that provides functionality specific for user „B“.

For simplicity, let us assume that the system can distinguish only two users („A“ and „B“) and each of the user-specific components *workerA* and *workerB* has only one provided interface and implements a simple storage of

5.3 An Example of a Component-Based System and its Description 65

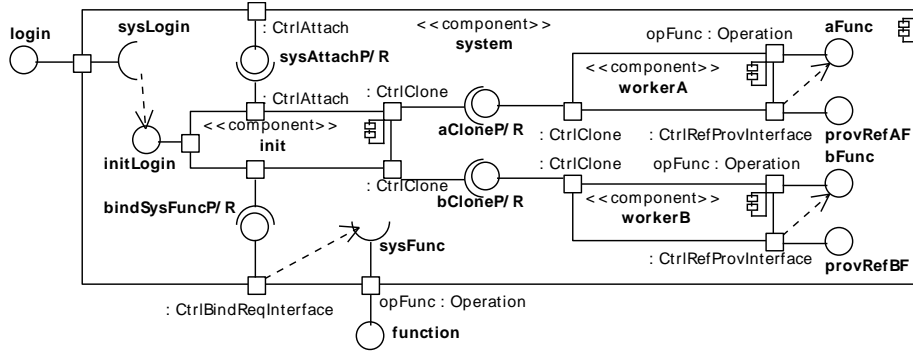


Fig. 5.8. The example of a simple component-based system that dynamically changes its behaviour, component `system` and its subcomponents `init`, `workerA`, and `workerB` (an initial configuration, i.e. without bound interface `sysFunc`).

one variable. The interface accepts an input value that will be stored in a component where it will replace a previously stored value returned via the interface as a response. Components `workerA` and `workerB` differ in the value returned on the first call.

The component-based system from the example is described in Figure 5.8 as follows. After the user’s credentials received by component `init` via its interface `initLogin` pass the verification, a clone of component `workerA` or `workerB` is acquired by component `init` via its required interface `aCloneR` or `bCloneR`, respectively. The new component is attached as a subcomponent of composite component `system` by component `init` via its required interface `sysAttachR`. Provided interface `aFunc` or `bFunc` of the new component `workerA` or `workerB`, respectively, is bound to required internal interface `sysFunc` of composite component `system` (i.e. a reference to `aFunc` or `bFunc` is acquired by `provRefAF` or `provRefBF` of the new component, respectively, and sent to provided internal interface `bindSysFuncP` of composite component `system`).

5.3.1 Definition of the Components’ Implementations

At first, we describe behaviour of cores of primitive components, i.e. the components’ implementations that have to be defined by developers of the component-based system from the example (see Section 5.2.5). Description of behaviour of component `workerA` and `workerB` is¹¹:

$$\begin{aligned}
 workerA_{core} &\stackrel{def}{=} (p_{aFunc}).workerA'_{core}[\text{undefA}, p_{aFunc}] \\
 workerA'_{core} &\stackrel{\Delta}{=} (val, p_{aFunc}).(p_{aFunc}(val', ret) \\
 &\quad \cdot (workerA'_{core}[val', p_{aFunc}] \mid \overline{ret}(val)))
 \end{aligned}$$

¹¹ The notation of π -calculus processes describing components has been defined in Section 5.2.1.

66 5 Component Model

$$\begin{aligned} workerB_{core} &\stackrel{def}{=} (p_{bFunc}).workerB'_{core}[undefB, p_{bFunc}] \\ workerB'_{core} &\triangleq (val, p_{bFunc}).(p_{bFunc}(val', ret) \\ &\quad .(workerB'_{core}[val', p_{bFunc}] | \overline{ret}\langle val \rangle)) \end{aligned}$$

where processes $workerA'_{core}$ and $workerB'_{core}$ can save a message received via name p_{aFunc} and p_{bFunc} , respectively, as name val' and send a previously saved message val as a reply via name ret . Names $undefA$ and $undefB$ represent initial values stored in components $workerA$ and $workerB$, respectively.

Behaviour of the $init$ component's core can be described as process abstraction $init_{core}$ with name $p_{initLogin}$ representing a provided functional interface, $r_{sysAttach}$ as a required interface to attach new subcomponents into the system (see Section 5.2.3), r_{aClone} and r_{bClone} as required interfaces for cloning components $workerA$ or $workerB$ (see Section 5.2.4), $r_{sysFunc}^s$ as a required interface for binding provided functional interfaces of the cloned components to a required functional interface of component $system$ represented by $sysFunc$ (see Section 5.2.2), and auxiliary name d . The behaviour is described as follows:

$$\begin{aligned} init_{core} &\stackrel{def}{=} (p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d). \\ &\quad p_{initLogin}(username, password). \\ &\quad (ok_A, ok_B, fail)(init_{verify}(username, password, ok_A, ok_B, fail) \\ &\quad + ok_A.init'_{core}[p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d] \\ &\quad + ok_B.init''_{core}[p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d] \\ &\quad + fail.init'''_{core}[p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d]) \\ init'_{core} &\triangleq (p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d).(new, d', t) \\ &\quad (\overline{r_{aClone}}\langle new \rangle.new(s'_0, s'_1, c', r', p').\overline{r_{sysAttach}}\langle s'_0, s'_1, d' \rangle. \\ &\quad p'(p'_{aFunc}. \overline{p'_{aFunc}}(t).t(aFunc').\overline{r_{sysFunc}^s}\langle aFunc' \rangle. \\ &\quad init_{core}\langle p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d' \rangle) | \overline{d} \\ init''_{core} &\triangleq (p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d).(new, d', t) \\ &\quad (\overline{r_{bClone}}\langle new \rangle.new(s'_0, s'_1, c', r', p').\overline{r_{sysAttach}}\langle s'_0, s'_1, d' \rangle. \\ &\quad p'(p'_{bFunc}. \overline{p'_{bFunc}}(t).t(bFunc').\overline{r_{sysFunc}^s}\langle bFunc' \rangle. \\ &\quad init_{core}\langle p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d' \rangle) | \overline{d} \\ init'''_{core} &\triangleq (p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d). \\ &\quad init_{core}\langle p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d \rangle | \overline{d}.d \end{aligned}$$

where process abstraction $init_{core}$ receives an user's initial request via name $p_{initLogin}$ as a pair of names $(username, password)$ and after successful verification of the user's name and password via the pseudo-application of $init_{verify}$, the process continues either as the constant application of $init'_{core}$ or as the constant application of $init''_{core}$.

5.3 An Example of a Component-Based System and its Description 67

The constant application of $init'_{core}$ or $init''_{core}$ requests a process representing behaviour of a cloned user-specific component via name r_{aClone} or name r_{bClone} , respectively. As a result, it receives control interfaces as names s'_0, s'_1, c' and sends them to a process representing component **system** via name $r_{sysAttach}$ to attach the user-specific component as its subcomponent. It also binds the user-specific component's functional provided interface represented by name $aFunc'$ or name $bFunc'$ and obtained via name p_{aFunc}^g or name p_{bFunc}^g , respectively, to an internal interface of component **system** by means of name $r_{sysFunc}^s$. Concurrently with the attaching the user-specific component as the subcomponent of **system**, an old subcomponent is detached via name d ¹².

The pseudo-application of $init_{verify}(username, password, ok_A, ok_B, fail)$ represents behaviour of a user's authentication and authorisation process (e.g. defined as $init_{verify} \stackrel{def}{=} (\dots).\overline{ok_A}$ to authorise users to component workerA).

5.3.2 Description of the Component Based System

Now, we can describe behaviour of the individual components including their control parts, as well as behaviour of a composite component that represents the whole component-based system from the example. According to Section 5.2.5, complete behaviour of components **workerA** and **workerB** can be described as follows:

$$\begin{aligned} workerA &\stackrel{def}{=} (s_0, s_1, c, p_{aFunc}^g).(p_{aFunc}) \\ &\quad (Ctrl_{Ifs}\langle p_{aFunc}, p_{aFunc}^g \rangle \mid Ctrl_{clone}[c] \mid workerA_{core}\langle p_{aFunc} \rangle) \\ workerB &\stackrel{def}{=} (s_0, s_1, c, p_{bFunc}^g).(p_{bFunc}) \\ &\quad (Ctrl_{Ifs}\langle p_{bFunc}, p_{bFunc}^g \rangle \mid Ctrl_{clone}[c] \mid workerB_{core}\langle p_{bFunc} \rangle) \end{aligned}$$

Behaviour of component **init** has to be described differently from the others, because it uses required control interfaces represented by names $r_{sysAttach}$, r_{aClone} , r_{bClone} , and $r_{sysFunc}^s$, which can not be referenced (contrary to functional interfaces, see Section 5.1.1). This case can be compared with the description of *Update* process in Section 5.2.4. The behaviour of component **init** can be described as follows:

$$\begin{aligned} init &\stackrel{def}{=} (s_0, s_1, c, p_{initLogin}^g, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s).(p_{initLogin}, d) \\ &\quad (Ctrl_{Ifs}\langle p_{initLogin}, p_{initLogin}^g \rangle \mid Ctrl_{clone}[c] \\ &\quad \mid init_{core}\langle p_{initLogin}, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s, d \rangle \mid d) \end{aligned}$$

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

¹² In a case of unsuccessful verification of the user's credentials, an actual subcomponent is detached and a trivial response to future attempts to detach is prepared (see $\bar{d}.d$ in the second part of process constant $init'''_{core}$).

68 5 Component Model

$$\begin{aligned}
 system &\stackrel{def}{=} (s_0, s_1, c, p_{login}^g, p_{function}^g) \cdot \\
 &\quad (p_{login}, r_{sysLogin}, p_{sysLogin}^s, p_{function}, r_{sysFunc}, p_{sysFunc}^s, p_{sysAttach}) \\
 &\quad (Ctrl_{clone}[c] \mid Ctrl_{Ifs}\langle p_{login}, p_{login}^g \rangle \mid Ctrl_{Ifs}\langle r_{sysLogin}, p_{sysLogin}^s \rangle \\
 &\quad \mid Ctrl_{Ifs}\langle p_{function}, p_{function}^g \rangle \mid Ctrl_{Ifs}\langle r_{sysFunc}, p_{sysFunc}^s \rangle \\
 &\quad \mid Ctrl_{EI}\langle p_{login}, r_{sysLogin} \rangle \mid Ctrl_{EI}\langle p_{function}, r_{sysFunc} \rangle \\
 &\quad \mid Ctrl_{SS}\langle s_0, s_1, p_{sysAttach} \rangle \\
 &\quad \mid system'\langle p_{sysAttach}, p_{sysLogin}^s, p_{sysFunc}^s \rangle) \\
 system' &\stackrel{def}{=} (p_{sysAttach}, p_{sysLogin}^s, p_{sysFunc}^s) \cdot \\
 &\quad (r_{sysAttach}, p_{initLogin}^g, r_{sysFunc}^s, s_0^{init}, s_1^{init}, s_0^A, s_1^A, s_0^B, s_1^B, c', d') \\
 &\quad (r_{aClone}, p_{aClone}, r_{bClone}, p_{bClone}, p_{aFunc}^g, p_{bFunc}^g) \\
 &\quad (init\langle s_0^{init}, s_1^{init}, c', p_{initLogin}^g, r_{sysAttach}, r_{aClone}, r_{bClone}, r_{sysFunc}^s \rangle \\
 &\quad \mid \overline{p_{sysAttach}}\langle s_0^{init}, s_1^{init}, d' \rangle \\
 &\quad \mid workerA\langle s_0^A, s_1^A, p_{aClone}, p_{aFunc}^g \rangle \mid \overline{p_{sysAttach}}\langle s_0^A, s_1^A, d' \rangle \\
 &\quad \mid workerB\langle s_0^B, s_1^B, p_{bClone}, p_{bFunc}^g \rangle \mid \overline{p_{sysAttach}}\langle s_0^B, s_1^B, d' \rangle \\
 &\quad \mid Wire[r_{aClone}, p_{aClone}, d'] \mid Wire[r_{bClone}, p_{bClone}, d'] \\
 &\quad \mid Wire[r_{sysAttach}, p_{sysAttach}, d'] \mid Wire[r_{sysFunc}^s, p_{sysFunc}^s, d'] \\
 &\quad \mid \overline{p_{initLogin}^g}\langle t \rangle.t(p_{initLogin}).\overline{r_{sysLogin}^s}\langle p_{initLogin} \rangle)
 \end{aligned}$$

Process abstraction *system* describes processing names representing control interfaces by means of the pseudo-applications of process abstractions *Ctrl_{clone}* (see Section 5.2.4), *Ctrl_{Ifs}* (see Section 5.2.2), *Ctrl_{EI}* (see Section 5.2.2), and *Ctrl_{SS}* (see Section 5.2.3). As process abstraction *system'*, it also creates concurrent pseudo-applications of process abstractions *init*, *workerA*, and *workerB* representing components *init*, *workerA*, and *workerB*, respectively, attaches them to the system via name *p_{sysAttach}*, interconnects names of required and provided control interfaces by means of the three constant applications of process constant *Wire*. Finally, it creates binding of name *p_{initLogin}* from process abstraction *init* representing component *init* to name *r_{sysLogin}* representing internal interface *sysLogin* of component *system* by means of names *p_{initLogin}^g* and *r_{sysLogin}^s*.

6

Behavioural Modelling of Services

This chapter deals with linking individual services of service-oriented architecture (SOA, see Chapter 4) 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 Chapter 5. The approach builds on a description of SOA and on its relation to component-based systems (CBSs, see Section 3.2) from Section 4.3.

According to the three levels of SOA abstraction from Section 4.1 and with respect to features of entities in SOA and CBSs (see Table 4.1 in the same section), 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 from Section 4.1). 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 [Er105]).
2. The service can be implemented as *a component-based system* (the „components“ level of abstraction from Section 4.1). 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, with respect to the flat model of SOA (see Section 4.1.3). Formal description of services according to the first view is introduced in Section 6.1.

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 Chapter 5. This approach is clarified in Section 6.2.

Finally, in Section 6.3, the approach is illustrated on an exemplary business process, specification of its services and description of their behaviour in the context of SOA and CBSs.

6.1 Service as a Part of Service Oriented Architecture

The result of business-to-service transformation [RW08], which forms SOA services from business processes (see Section 4.1.2), 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*» (for an example, see Figure 6.3). 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, in the terminology of Section 4.1).

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 from Section 4.1 and in the first view according to the introduction of Chapter 6). Behaviour of the service can be described as π -calculus process abstraction *Service* as follows:

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

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

6.1.1 Communication of Services and Service Broker

Communication of services in SOA is realised by means of various styles of passing data (see Section 4.2). In a case of existing *service choreography or orchestration* in SOA (see Section 4.1), 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 [Erl05]. To support the dynamic service discovery and invocation, SOA provides service brokers (e.g. UDDI registries, see [CHvRR04]), 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:

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

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 $\text{Publish}[t', pub]$, which handles future requests, and process $\bar{t}\langle t', i, d \rangle$, which enqueues the received pair (i, d) by sending them via name t , that represents *the current tail of the queue*, together with name t' , that represents a new tail of the queue used in the future requests.

The constant application of *Find* dequeues a front item of the queue as a triple of names (h', i, d) via name h , that represents *the current head of the queue*. Then, it proceeds as a composition of a constant application of $\text{Find}[h', find, pub]$, 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.

6.2 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 from Section 4.1 and the second view from the introduction of Chapter 6.

To describe a service **Service** with interfaces **I1** to **In** as a component-based system and by means of the component model from this book (see Chapter 5), we need to transform π -calculus process abstraction *Service* from Section 6.1 describing behaviour of the service into a formal description of behaviour of a component representing the component-based system (see Section 5.2). We focus on pseudo-application $Svc_j\langle i_j, b_1, \dots, b_m \rangle$, which describes specific processing of the service’s interface i_j (for $j \in \{1, \dots, n\}$) and communication with other parts of the service via shared names b_1, \dots, b_m . Process abstraction Svc_j can be defined as

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

where $k \in \{1, \dots, m\}$ and $x_1, \dots, x_k, y_1, \dots, y_{(m-k)} \in \{1, \dots, m\}$, and sets $\{b_{x_1}, \dots, b_{x_k}\} \cap \{b_{y_1}, \dots, b_{y_{(m-k)}}\} = \emptyset$ and $\{b_{x_1}, \dots, b_{x_k}\} \cup \{b_{y_1}, \dots, b_{y_{(m-k)}}\} = \{b_1, \dots, b_m\}$ (see the pseudo-application of Svc_j in Section 6.1).

Name i_j represents the interface **Ij** provided by the service, names b_{x_1}, \dots, b_{x_k} are all of the shared names that are used as channels of input prefixes in Svc'_j and names $b_{y_1}, \dots, b_{y_{(m-k)}}$ are all of the shared names that are used as channels of output prefixes in Svc'_j (for input and output prefixes, see Section 2.3). Thereafter, process abstraction Svc'_j can be understood as a description of core behaviour of a component with functional provided interfaces represented by names $i_j, b_{x_1}, \dots, b_{x_k}$ and functional required interfaces represented by names $b_{y_1}, \dots, b_{y_{(m-k)}}$ in the external view (see Section 5.2).

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:

$$\begin{aligned} Svc'_j \stackrel{def}{=} & (i_j, b_{x_1}, \dots, b_{x_k}, b_{y_1}, \dots, b_{y_{(m-k)}}). \\ & (s_0, s_1, c, p_1^s, \dots, p_{(m-k)}^s, p_1^g, \dots, p_{(k+1)}^g) \\ & \left(\prod_{u=1}^k (d, t) \overline{(p_{(u+1)}^g)}(t).t(p).Wire[b_{x_u}, p, d] \mid \prod_{v=1}^{m-k} \overline{p_v^s}(b_{y_v}) \right) \\ & \mid (d, t) \overline{(p_1^g)}(t).t(p).Wire[i_j, p, d] \\ & \mid Comp_j\langle s_0, s_1, c, p_1^s, \dots, p_{(m-k)}^s, p_1^g, \dots, p_{(k+1)}^g \rangle \end{aligned}$$

where process constant *Wire* has been defined in Section 5.2.2 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 5.2.5.

6.3 An Example of a Service-Oriented Architecture

In this section, the approach proposed in Section 6.1 and Section 6.2 is illustrated on an exemplary business process of „Process Purchase Order“, which is adopted from [OMG06b].

The business process is described in a *business process model* (BPM, see Section 4.1.1) in Figure 6.1. There are three categories of activities that are responsible for realisation of the „Process Purchase Order“: „Invoicing“, „Shipping“, and „Scheduling“. Processing starts by receiving a purchase order message. Afterwards, the „Invoicing“ activities calculate an initial price. This price is not yet complete, because the total price depends on where the products are produced and on the amount of the shipping cost. In parallel, the „Shipping“ activities determine when the products will be available and from what locations. After the shipping information is known, the complete price can be calculated. At the same time, the process requests a shipping schedule from the „Scheduling“ activities. Finally, when the complete price, the shipping info and the shipping schedule are available, the invoice can be completed and sent to the customer.

6.3.1 Service Identification

In the first step of *service identification* (see Section 4.1.2), the following tasks from the BPM can be identified as invocations of services: „Initiate Price Calculation“, „Complete Price Calculation“, „Request Shipping“, „Request Production Scheduling“, and „Send Shipping Schedule“.

Figure 6.2 shows an UML component diagram [OMG07b] of the services in the „lollipop“ style notation (see Section 3.4.2). Classes stereotyped as «service» represent five business (entity) services `InitPriceCalculator`, `CompletePriceCalculator`, `Shipping`, `ProductionScheduling`, and `ShippingScheduling`, and two controller (task-centric) services `PurchaseOrderProcessing` and `Scheduling`.

The *business services* are derived according to the service invocation tasks and provide functional capabilities defined by the tasks, while the *controller services* are access points to orchestrations of another business or controller services (see Section 4.1). Service `PurchaseOrderProcessing` represents the whole business process of „Process Purchase Order“ from Figure 6.1. It orchestrates services `InitPriceCalculator`, `CompletePriceCalculator`, `Shipping`, and `Scheduling`. Service `Scheduling` orchestrates services `ProductionScheduling` and `ShippingScheduling`.

Service providers and service consumers are distinguished in Figure 6.2 by means of provider and consumer *ports*, respectively, as it has been introduced in [RW08]. Each class representing a service realises at least one interface at its provider port, which describes a functional capability provided by the service. Moreover, services `PurchaseOrderProcessing`, `CompletePriceCalculator`, `Shipping`, and `Scheduling` can be *invoked asynchronously*. After

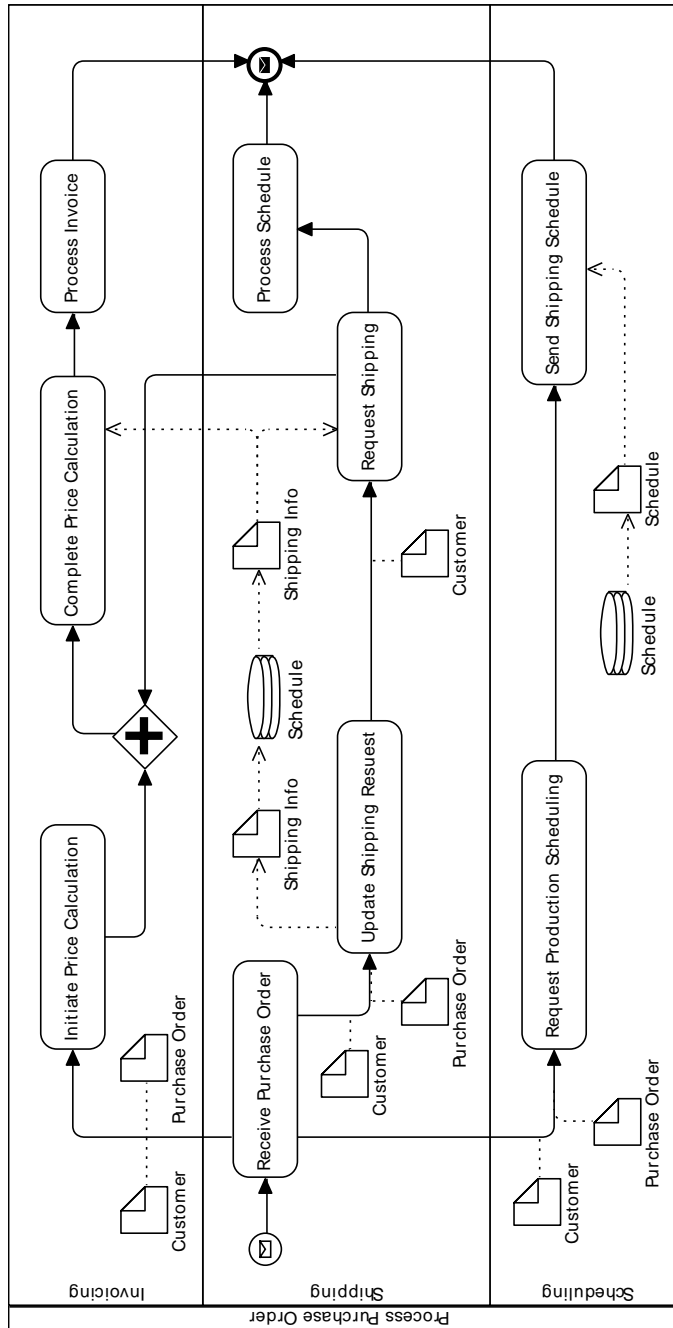


Fig. 6.1. Business process model of “Process Purchase Order” (adopted from [OMG06b]).

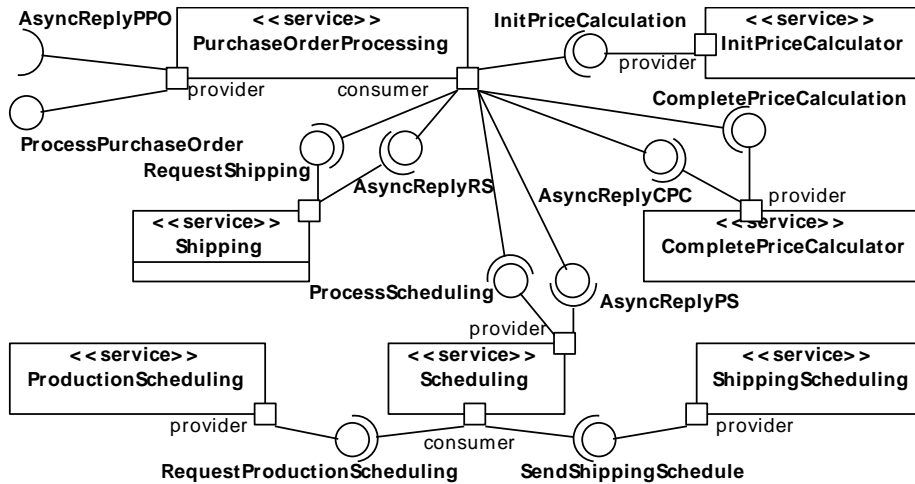


Fig. 6.2. An overview of identified services and their interconnections.

their asynchronous invocations, specific responses can be obtained via interfaces `AsyncReplyPPO`, `AsyncReplyPPC`, `AsyncReplyRS`, and `AsyncReplyPS`, respectively. These interfaces are provided at consumer ports of classes that represent replying services (i.e. the asynchronously invoked services), instead of at their provider ports, to distinguish them from regular service invocations.

6.3.2 Service Model

To provide an example of behavioural description of SOA, we focus on controller service `Scheduling`, which orchestrates `ProductionScheduling` and `ShippingScheduling`. Detailed description of these services as classes with stereotype `«service»` and their interfaces with stereotype `«interface»` is provided in the UML class diagram in Figure 6.3. The relationships between services and interfaces are stereotyped as `«use»` (the services require the interfaces) or they are UML relations of „realisation“ (the services provide the interfaces).

Service `Scheduling` is invoked asynchronously with parameters describing `customer`, `purchaseOrder`, an identification of the request (`requestID`), and a service that will accept a reply (`replyToURL`). Its behaviour is described by means of an UML sequence diagram in Figure 6.4 as a sequence of service invocations.

After receiving an external request from a consumer, service `PurchaseOrderProcessing` asynchronously calls its orchestrated services, including controller service `Scheduling`. Then, service `Scheduling` synchronously calls its first orchestrated service `ProductionScheduling`, its second orchestrated service `ShippingScheduling` and finally, it notifies service `PurchaseOrder-`

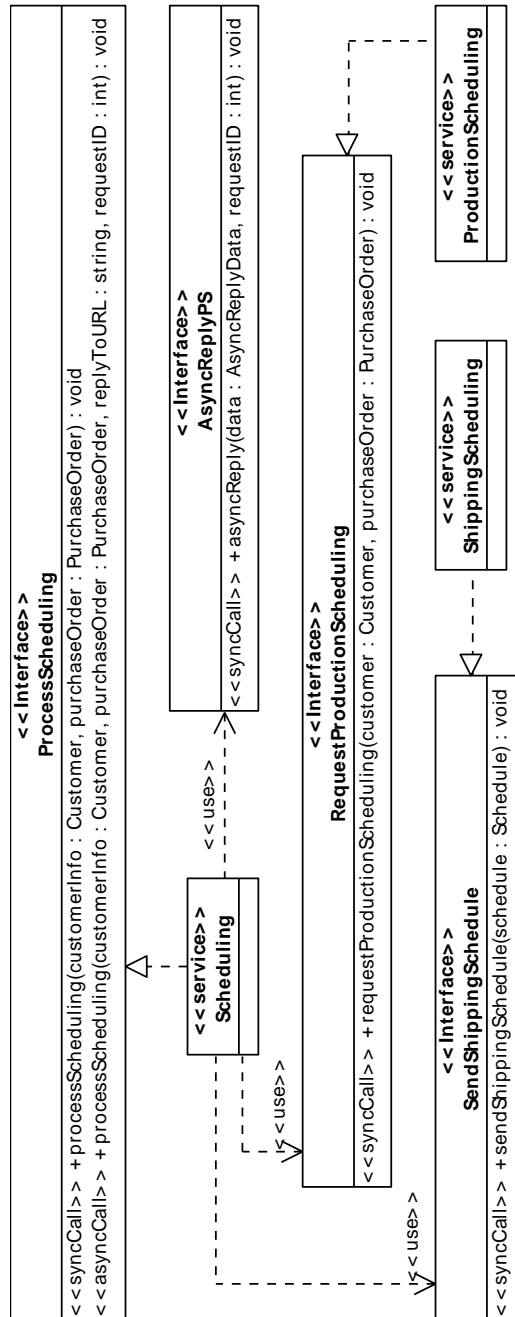


Fig. 6.3. Controller service Scheduling and its orchestration of business services ProductionScheduling and ShippingScheduling.

6.3 An Example of a Service-Oriented Architecture 77

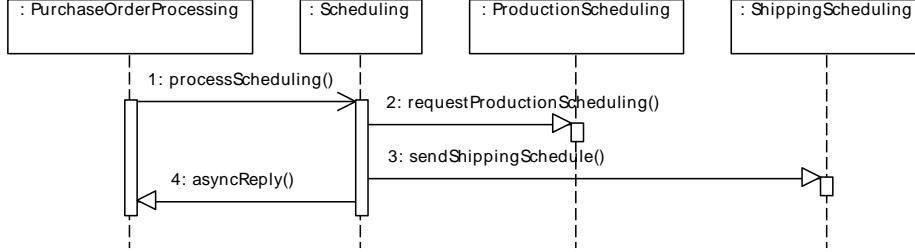


Fig. 6.4. Behaviour of service Scheduling as a sequence of service invocations.

Processing via its interface AsyncReplyPS that the processing has been finished.

6.3.3 Description of Services as Entities of SOA

Now, we are ready to describe behaviour of services Scheduling, ProductionScheduling, and ShippingScheduling as entities of SOA by means of π -calculus process abstractions S , PS , and SS , respectively (see Section 6.1). These process abstractions use names ps , rps , and sss as representations of the services’ interfaces ProcessScheduling, RequestProductionScheduling, and SendShippingSchedule, respectively. The process abstractions are defined as follows:

$$\begin{aligned}
 S &\stackrel{def}{=} (ps, get_{rps}, get_{sss}).(rps, sss) \\
 &\quad (S_{init}\langle get_{rps}, get_{sss}, rps, sss \rangle.S_{impl}[ps, rps, sss]) \\
 S_{init} &\stackrel{def}{=} (get_{rps}, get_{sss}, rps, sss).get_{rps}(rps).get_{sss}(sss) \\
 PS &\stackrel{def}{=} (rps, set_{rps}).(d)(PS_{init}\langle rps, set_{rps}, d \rangle.PS_{impl}[rps]) \\
 PS_{init} &\stackrel{def}{=} (rps, set_{rps}, d).\overline{set_{rps}}\langle rps, d \rangle \\
 SS &\stackrel{def}{=} (sss, set_{sss}).(d)(SS_{init}\langle sss, set_{sss}, d \rangle.SS_{impl}[sss]) \\
 SS_{init} &\stackrel{def}{=} (sss, set_{sss}, d).\overline{set_{sss}}\langle sss, d \rangle
 \end{aligned}$$

where ps , rps , and sss are the names representing the interfaces and subsequently processed by applications of process constants S_{impl} , PS_{impl} , and SS_{impl} , respectively. Initialisation of the services is described by means of process abstractions S_{init} , PS_{init} , and SS_{init} , which are applied before the mentioned process constants. The process abstractions use names get_{rps} , set_{rps} , get_{sss} , and set_{sss} as connections to process abstractions describing behaviour of specific service brokers. The brokers allow to store and retrieve references to interfaces RequestProductionScheduling and SendShippingSchedule of services ProductionScheduling and ShippingScheduling, respectively.

Finally, complete service-oriented architecture of communicating services and brokers can be described as process abstraction *System* as follows:

$$\begin{aligned} System &\stackrel{def}{=} (ps).(get_{rps}, set_{rps}, get_{sss}, set_{sss}) \\ &\quad (S\langle ps, get_{rps}, get_{sss} \rangle \mid (rps)PS\langle rps, set_{rps} \rangle \mid (sss)SS\langle sss, set_{sss} \rangle \\ &\quad \mid Broker\langle set_{rps}, get_{rps} \rangle \mid Broker\langle set_{sss}, get_{sss} \rangle) \end{aligned}$$

For testing purposes (e.g. to verify an interoperability of the services), we may need to finish π -calculus description of process constants S_{impl} , PS_{impl} , and SS_{impl} . These process constants describe internal behaviour of the individual services, which will be defined later as behavioural description of underlying component-based systems in Section 6.3.4. However, without additional knowledge of the services’ underlying implementation, we can describe their communication behaviour as follows:

$$\begin{aligned} S_{impl} &\stackrel{\Delta}{=} (ps, rps, sss).(r_{rps}, r_{sss}, s) \\ &\quad (ps\langle ci, po, r_{ps} \rangle.\overline{r_{ps}}\langle ci, po, r_{rps} \rangle.r_{rps}.\overline{sss}\langle s, r_{sss} \rangle.r_{sss}.\overline{r_{ps}}) \\ PS_{impl} &\stackrel{\Delta}{=} (rps).r_{ps}\langle ci, po, r_{rps} \rangle.\overline{r_{rps}} \\ SS_{impl} &\stackrel{\Delta}{=} (sss).sss\langle s, r_{sss} \rangle.\overline{r_{sss}} \end{aligned}$$

where the process constants describe the sequences of service invocations according to the sequence diagram in Figure 6.4, names *ci*, *po*, and *s* represent parameters **customer**, **purchaseOrder**, and **schedule**, respectively, of methods **processScheduling**, **requestProductionScheduling**, and **sendShippingSchedule** in the specific interfaces according to the class diagram in Figure 6.3.

6.3.4 Description of Services as Component-Based Systems

This section deals with description of service **Scheduling**, which is a controller service of business services **ProductionScheduling** and **ShippingScheduling**, as an underlying component-based system. The system is represented as a component of the component model from Chapter 5. We continue in the service’s behavioural description from Section 6.3.3 as it has been proposed in Section 6.2.

Constant application $S_{impl}[ps, rps, sss]$ has been used in process abstraction S in Section 6.3.3 to represent behaviour of the component that implements service **Scheduling**. It contains name *ps*, which represents an interface provided by service **Scheduling**, and names *rps* and *sss*, which represent interfaces required by the service from services **ProductionScheduling** and **ShippingScheduling**, respectively.

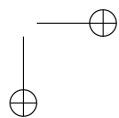
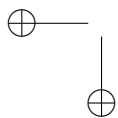
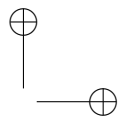
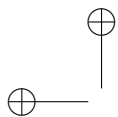
Process constant S_{impl} can be defined as follows:

6.3 An Example of a Service-Oriented Architecture 79

$$\begin{aligned}
 S_{impl} \stackrel{def}{=} & (ps, rps, sss).(s_0, s_1, c, rps^s, sss^s, ps^g) \\
 & (\overline{rps^s}\langle rps \rangle \mid \overline{sss^s}\langle sss \rangle \mid (d, t)(\overline{ps^g}\langle t \rangle.t(p).Wire[ps, p, d]) \\
 & \mid S_{comp}(s_0, s_1, c, rps^s, sss^s, ps^g))
 \end{aligned}$$

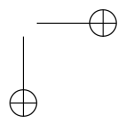
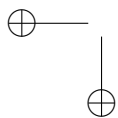
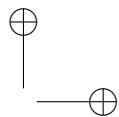
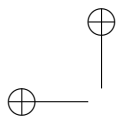
where pseudo-application of S_{comp} describes behaviour of the component that implements service **Scheduling**, names s_0 and s_1 represent control interfaces of the component’s life-cycle, name c represents an interface for its cloning, name ps^g represents a control interface for referencing the component’s provided functional interface and names rps^s and sss^s represent control interfaces for binding the component’s required functional interfaces.

Definition of process abstraction S_{comp} , as well as process abstractions PS_{comp} and SS_{comp} , can be derived from models of component-based systems, which realise individual services **Scheduling**, **ProductionScheduling** and **ShippingScheduling**, respectively, as it has been proposed in Section 5.2.



Part III

Application



7

Development Process

In this chapter, we propose an application of the component model from Chapter 5 and the behavioural modelling of services from Chapter 6 in a software development process. The development process should conform to principles of the component-based development (CBD, see Section 3.2) and should incorporate service-oriented architecture (SOA, see Chapter 4).

The behavioural modelling of services and the component model can be applied in a design phase of the development process as it is described in Section 7.1 and Section 7.2, respectively. Section 7.3 briefly outlines integration of a formal description resulting from the behavioural modelling into the development process.

7.1 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.

Service identification has been described in Section 4.1.2 and demonstrated in the example in Section 6.3.1.

2. The candidate services and their interfaces¹ 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

¹ Services that invoke asynchronously other services have to implement extra auxiliary interfaces for receiving asynchronous replies.

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 [RW08], where possible interconnections of the services will be modelled². 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.

Modelling of services has been described in Sections 4.1.2 and 4.1.3 and demonstrated in the example in Section 6.3.2.

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“).

Behavioural modelling of services in the π -calculus has been described in Chapter 6 and demonstrated in the example in Section 6.3.3.

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 7.2).

7.2 Application of the Component Model

A component-based system can be modelled in a logical view (structural description, see Section 5.1) and in a process view (behavioural description, see Section 5.2). 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.

² However, the component diagram may be created during the service identification, as it has been described in Section 6.3.1.

7.2 Application of the Component Model 85

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.

Structural modelling of component-based systems has been described in Section 5.1 and demonstrated in the example in Section 5.3.

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 behaviour³ 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.

Formal description of *behaviour of primitive components* has been introduced in Section 5.2 and demonstrated in the example in Section 5.3.

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.

Behaviour of composite components has been formally described in Section 5.2 and demonstrated in the example in Section 5.3.

³ The core behaviour describes processing of functional interfaces and custom control interfaces only (see process abstraction $Comp_{impl}$ in Section 5.2.5).

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 7.1) realised as underlying component-based systems, each service will be modelled and its behaviour described separately as a stand-alone component-based system.

7.3 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 (see Definition 8 in Section 2.3) without any special extensions, several existing tools can be used for model checking of resulting π -calculus processes and formal verification of their properties. Some of the tools will be introduced in Section 8.2 and their applications will be demonstrated in Section 9.6.

8

Tools

This chapter deals with software tools that support the proposed component model from Chapter 5. The tools can be divided into two groups: component modelling tools and verification tools, which are described in Section 8.1 and Section 8.2, respectively.

The *component modelling tools* provide developers with ability to design models of component-based systems (i.e. the system models from Section 5.1.2), while the *verification tools* allow to simulate and analyse formally described behaviour of the systems (i.e. the π -calculus processes from Section 5.2 and Chapter 6).

8.1 Component Modelling Tools

Component modelling tools provide a supporting environment for integration of the component model into software development processes.

8.1.1 Component Diagrams in UML

According to Section 5.1.2, models of component-based systems can be described as specific component diagrams. Their notation is based on UML component diagrams where it utilises a „dependency“ relation (see Section 3.4.2), which represents bindings of required functional interfaces to provided functional interfaces, as well as connections between components' control interfaces of special types and related functional interfaces or another components¹. Therefore, the specific component diagrams are compatible with standard

¹ The special types are namely, `CtrlBindReqInterface`, `CtrlPrefProvInterface`, `RefToProvInterface`, and `RefToComponent`, of control interfaces for binding required functional interfaces, for referencing provided functional interfaces, for passing references to provided functional interfaces, and for passing references to components, respectively (see Section 5.1.1).








UML component diagrams [OMG07b] and the component-based systems can be modelled in common UML modeling tools (e.g. *Visual Paradigm for UML* or *Poseidon for UML*²).

However, these modeling tools do not respect semantics associated with the component model’s entities and can not check its constraints. A developer may be allowed to design models of component-based systems that do not implement the component model’s metamodel (see Section 5.1.1).

8.1.2 A Tool for Modelling of Component-Based Systems

To design component-based systems according to the component model’s metamodel, a *tool for modelling of component-based systems* has been developed. The tool is a product of master’s thesis [Gal09], which has been supervised by one of the authors of this book. It uses *Eclipse Modeling Framework* (EMF) [BSM⁺03, Ecl07b] and *Eclipse Graphical Modeling Framework* (GMF) [Ecl07a] to provide a graphical editor of models of component-based systems. At the time of writing this book, a stable version of the tool is based on a metamodel that has been published in [Ryc09]. The metamodel is a preliminary and different version of the component model’s metamodel from Section 5.1.1.

The metamodel, which is used in the tool, is described in Figure 8.1 as an Eclipse Ecore diagram designed for EMF. It distinguishes between a binding of two interfaces that are at the same level of a component hierarchy (class `Connection`) and a binding of a composite component’s external interfaces to its internal interfaces and vice versa (class `InternalComponentProxy`)³. As the preliminary version, the proposed metamodel lacks many extended features of the component model’s metamodel, such as typed functional interfaces or the control interfaces.

Figure 8.2 shows a model of the component-based system from Section 5.3, which has been created by means of the tool for modelling of component-based systems (a reduced version of the original model without control interfaces). Composite components are represented by instances of `CompositeComponent` from the metamodel and denoted by icon , primitive components are instances of `PrimitiveComponent` denoted by icon , required interfaces are instances of `RequiredInterface` denoted by icon , and provided interfaces are instances of `ProvidedInterface` denoted by icon . Interfaces are interconnected by means of connections, which are instances of `Connection` from the metamodel and denoted by grey arrows and icon . Grey arrows and icons  or  denote bindings of internal required or provided interfaces of a composite component to its external provided or required

² See <http://www.visual-paradigm.com/product/vpuml/> and <http://www.gentleware.com/products.html>, respectively.

³ In the component model’s metamodel from Section 5.1.1, these bindings are unified in class `Binding` and its different realisations (see Figure 5.4)

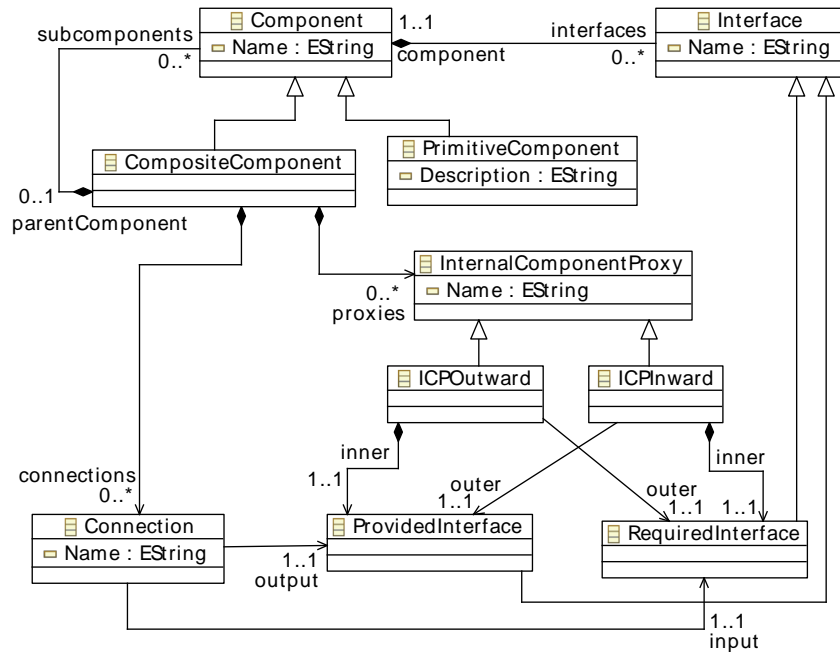


Fig. 8.1. Eclipse Ecore diagram of the metamodel, which is used in the tool for modelling of component-based systems (adopted from [Gal09], a full version can be found in [Ryc09]).

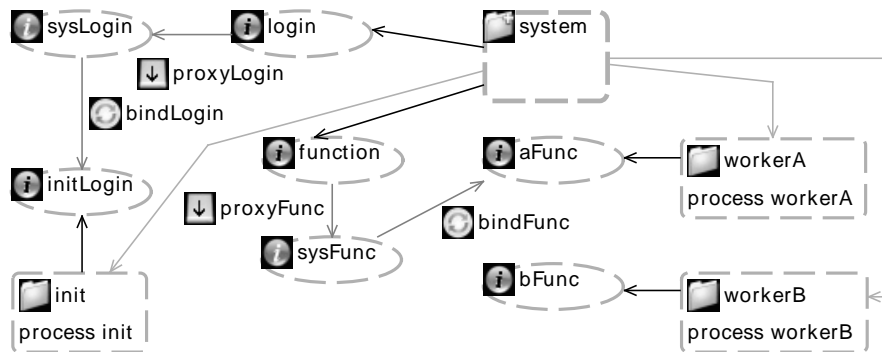


Fig. 8.2. The model of the component-based system from the example in Section 5.3 (adopted from Figure 5.8) with component `system` and its subcomponents `init`, `workerA`, and `workerB`, without control interfaces.

interfaces, respectively, as instances of `ICPInward` or `ICPOutward` from the metamodel. Relations of composite components and their internal interfaces or subcomponents are denoted by black or grey arrows, respectively (see the relation of `CompositeComponent` as a parent component and `Component` as its subcomponent or the relation of `CompositeComponent` and `Interface` in the metamodel).

Composite component system consists of its subcomponents `init`, `workerA`, and `workerB`⁴ and provides external interfaces `login` and `function`. Requests arriving at these interfaces are forwarded inside the composite component by means of `proxyLogin` and `proxyFunc` and by means of its internal required interfaces `sysLogin` and `sysFunc`, respectively. The internal interfaces are bound via `bindLogin` and `bindFunc` to provided interfaces `initLogin` and `aFunc` of components `init` and `workerA`, respectively (provided interface `bFunc` of component `workerB` is not bound in this configuration). Behaviour of the subcomponents has been described by means of process abstractions „`init`“, „`workerA`“ and „`workerB`“ in Section 5.3.

In the tool, the model of the component-based system can be stored in *XML Metadata Interchange* (XMI) format, which is an *Object Management Group’s* (OMG) standard for exchanging metadata information in XML and which is commonly used as an interchange format for MOF-based models (e.g. UML models; see Section 5.1). Listing 8.1 describes the model of the component-based system from Figure 8.2 in XMI format.

8.2 Verification Tools

Behaviour of a component-based system designed according to the component model from Chapter 5 can be described formally as a π -calculus process (see Section 5.2). The formally described behaviour can be verified by means of external verification tools.

In this section, we describe the verification tools of polyadic π -calculus processes. Section 8.2.1 deals with *The Mobility Workbench* (MWB, [Vic95]), a model checker and an open bisimulation checker of mobile concurrent systems described in the π -calculus. In Section 8.2.2, we describe the *Another/Advanced Bisimulation Checker* (ABC, [Bri05]), which allows to check open-equivalences in the π -calculus, in a similar way as MWB but with some improvements. Finally, in Section 8.2.3, the *Pi-Calculus Equivalences Tester* (PiET, [Mio06]) is described, which 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.

The open bisimulation checkers, i.e. MWB and ABC, will be used later in Section 9.6. For the theoretical background and further references, see Section 2.3.2 and [SW03].

⁴ The tool connects subcomponents and their parent component via relations denoted by grey arrows.


```

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:diagram2="http://diagram/1.0">
<diagram2:CompositeComponent/>
<diagram2:CompositeComponent Name="system">
  <interfaces xsi:type="diagram2:ProvidedInterface"
    Name="login"/>
  <interfaces xsi:type="diagram2:ProvidedInterface"
    Name="function"/>
  <subcomponents
    xsi:type="diagram2:PrimitiveComponent"
    Name="init" Description="process_init">
    <interfaces
      xsi:type="diagram2:ProvidedInterface"
      Name="initLogin"/>
    </subcomponents>
  <subcomponents
    xsi:type="diagram2:PrimitiveComponent"
    Name="workerA" Description="process_workerA">
    <interfaces
      xsi:type="diagram2:ProvidedInterface"
      Name="aFunc"/>
    </subcomponents>
  <subcomponents
    xsi:type="diagram2:PrimitiveComponent"
    Name="workerB" Description="process_workerB">
    <interfaces
      xsi:type="diagram2:ProvidedInterface"
      Name="bFunc"/>
    </subcomponents>
  <connections Name="bindLogin"
    output="/1/@subcomponents.0/@interfaces.0"
    input="/1/@proxies.0/@inner"/>
  <connections Name="bindFunc"
    output="/1/@subcomponents.1/@interfaces.0"
    input="/1/@proxies.1/@inner"/>
  <proxies xsi:type="diagram2:ICPInward"
    Name="proxyLogin" outer="/1/@interfaces.0">
    <inner Name="sysLogin"/>
  </proxies>
  <proxies xsi:type="diagram2:ICPInward"
    Name="proxyFunc" outer="/1/@interfaces.1">
    <inner Name="sysFunc"/>
  </proxies>
</diagram2:CompositeComponent>
<diagram2:ProvidedInterface Name="test"/>
</xmi:XMI>

```

Listing 8.1. The model of a component-based system from Section 8.1.2 stored in XML, the OMG standard XML metadata interchange format.

8.2.1 The Mobility Workbench (MWB)

*The Mobility Workbench*⁵ (MWB, [Vic95]) is a tool for open bisimulation checking, model checking, finding deadlocks, and interactive simulation of mobile concurrent systems described in the π -calculus. It has been developed by Bjorn Victor, Faron Moller, Lars-Henrik Eriksson, and Mads Dam in functional programming language Standard ML (SML), a dialect of Robin Milner’s ML programming language, for its compiler Standard ML of New Jersey⁶ (SML/NJ).

The tool is based on an algorithm published in [Vic94], which allows to decide the open (strong and weak) bisimulation equivalences (see Section 2.3.2) for agents in the polyadic π -calculus possible containing an positive match operator⁷ [Vic95].

In MWB, π -calculus process abstractions are represented by agents and described by means of the π -calculus grammar with modified syntax: input prefix $x(m)$ is typed as $\mathbf{x}(m)$, output prefix $\bar{y}(m)$ is typed as $\mathbf{y}\langle m \rangle$, internal (silent) action τ is typed as \mathbf{t} , restriction $(z)P$ is typed as $(\mathbf{z})P$, abstraction $P \triangleq (a_1, \dots, a_n) \dots$ is typed as $P = (\mathbf{a}1, \dots, \mathbf{a}n) \dots$, and application $P[a_1, \dots, a_n]$ is typed as $P(\mathbf{a}1, \dots, \mathbf{a}n)$ ⁸.

The agents have to be closed, i.e. their free names must be a subset of their argument lists [Vic95]. Moreover, recursively defined agents can not be reduced by their applications without any input actions, output actions or internal (silent) actions (see Definitions 6 and 11 in Section 2.3), i.e. *only guarded recursions are handled correctly*. For this reason, MWB does not support constant applications of recursively defined π -calculus process constants that implement the *replication operator* (i.e. $!P$, see Definition 6 in Section 2.3).

Process abstractions and constants representing formal description of behaviour of service-oriented architectures and component-based systems can be transformed into MWB agents. In cases of the process abstractions or constants that use constant applications of recursively defined process constants to implement the replication operator, we have to bypass the limitation of MWB and insert a π -calculus prefix before each recursive constant application (see Definition 11 in Section 2.3). After this modification, the recursions will be guarded. The modification of the problematic process constants must not affect communication with their well-established environment (the process constants describe behaviour of specific services, components or their parts, which should not be affected by the modification). Therefore, the inserted

⁵ The Mobility Workbench can be found at <http://www.it.uu.se/research/group/mobility/mwb>.

⁶ For Standard ML of New Jersey, see <http://www.smlnj.org/>.

⁷ A process $[x = y]\pi.P$, which contains the *positive match operator* $[x = y]$, can evolve as $\pi.P$ iff x and y are the same name (see [SW03]).

⁸ The MWB does not distinguish a pseudo-abstraction/application from a constant abstraction/application.

prefix should be reducible by an internal (silent) action, i.e. it has to be an unobservable prefix τ , which directly meets the requirement, or an input or output prefix with an additional name, which meets the requirement after composition with an auxiliary process communicating via the same name.

The MWB can be used, for example:

- to verify that agents representing behaviour of two components are open bisimilar (see Definition 16 in Section 2.3.2), e.g. after an update of a component in a component-based system, when a proof of equivalence of original and updated behaviour is needed;
- to check if a specific agent contains deadlocks and to obtain their descriptions, which means that a component may not be ready to handle external requests from its environment;
- to trace possible reductions of a specific agent in a specific environment, i.e. to simulate each step of a component’s behaviour and its external communication, and to debug the component’s behaviour.

8.2.2 Another/Advanced Bisimulation Checker (ABC)

The *Another Bisimulation Checker*⁹, also referred as *Another Bisimilarity Checker* or *Advanced Bisimulation Checker* (ABC, [Bri05]), is a tool that checks for open bisimulation between terms of the π -calculus. The tool has been developed by Sébastien Briais and implemented in functional programming language Objective Caml¹⁰ (OCaml), an object-oriented extension of a dialect of Robin Milner’s ML programming language.

In comparison with MWB (see Section 8.2.1), the ABC does not implement model checking and finding deadlocks, but provides an user with improved equivalence checking [BN07] and interactive simulations. However, analogously to the limitations of MWB and due to the same theoretical reasons, the ABC does not support constant applications of recursively defined π -calculus process constants.

8.2.3 Pi-Calculus Equivalences Tester (PiET)

The *Pi-Calculus Equivalences Tester*¹¹ (PiET, [Mio06]) is a tool for checking of 10 different types of π -calculus equivalences: strong and weak ground equivalences, strong and weak early equivalences (which are equal to barbed equivalences), strong and weak early congruences (which are equal to ground congruences and to barbed congruences), strong and weak late equivalences, and strong and weak late congruences. The congruences corresponding to the

⁹ The Another Bisimulation Checker can be found at <http://lamp.epfl.ch/~sbriais/abc/>.

¹⁰ For Objective Caml, see <http://caml.inria.fr/ocaml/>.

¹¹ The Pi-Calculus Equivalences Tester can be found at <http://piet.sourceforge.net/>.

ground, early, and late equivalences are obtained with closure over all contexts [Mio06].

The tool has been developed by Matteo Mio, theoretically based on [Lin00] and implemented in functional programming language Fresh Objective Caml¹² (Fresh O’Caml), an extended version of the Objective Caml, with a graphical user interface implemented in Java.

In comparison with MWB and ABC (see Section 8.2.1 and Section 8.2.2, respectively), the PIET does not implement checking of open bisimulation, which is a finer relation than the late and early bisimulations [Lin00].

¹² For Fresh Objective Caml, see <http://www.fresh-ocaml.org/>.

9

Case Study

In this chapter, we will demonstrate an application of service-oriented architectures, the component model, and the behavioural description of component-based systems from Chapters 5 and 6. To validate our approach, we will adopt a case study of *a service-oriented architecture for functional testing of complex safety-critical systems* from [DMM⁺08]. It allows to distribute and run specific tests over a wide range of different testing environments, varying in their logical positions in a system’s architecture. We will proceed according to the development process that has been described in Chapter 7.

In the context of our approach, the case study has the following interesting features:

- the safety-critical systems usually „consist of many subcomponents which are tightly coupled and have highly complex interactions“ [BS93] – it is useful to describe a safety-critical system as a component-based system from Section 3.2;
- the tests are distributed to different parts of the system’s architecture, run in different contexts, and interact with their local testing environments – the architecture is evolving as the mobile architecture from Section 3.1;
- the original case study [DMM⁺08] describes the system’s architecture as a specific service-oriented architecture – we can use our approach from Chapter 6 to describe individual services as component-based systems;
- the functional testing process has been described and verified on existing testing environments of a railway interlocking control system in [DMM⁺08] – the provided case study is based on a real-world instance of a problem¹;

¹ Railway systems, in general, have been subjects of many formal approaches in recent years [BS93].

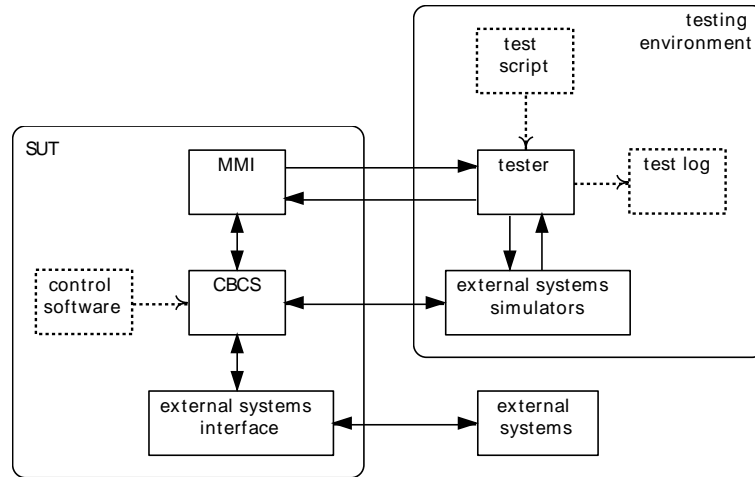


Fig. 9.1. Testing environment of a railway interlocking control system (adopted from [DMM⁺08]).

- the formal description and possible verification of functional testing in a complex safety-critical system can increase reliability of test results and contribute to the safety of the system².

9.1 System Description

A testing environment of a railway interlocking control system is described in Figure 9.1 and in [DMM⁺08] as a composition of a set of external system simulators and a tester. The *external system simulators* interact with a *system under testing* (SUT) and represent and simulate behaviour of its field objects (points, track circuits, coloured signals, etc.³). A *tester* automatically executes specific tests, which are coded in *test scripts*, and coordinates relevant external system simulators. It also interacts with SUT by means of its *man machine*

² The railway interlocking control systems, as well as their testing environments, have to be verified and certified as required by international standards.

³ Each *track circuit* detects, by means of *sensors*, a presence of trains in a specific section of a track and properly alters the *coloured signals* to reflect their presence or absence. According to the presence or absence of the individual trains in the individual tracks and their sections, the railway interlocking system switches specific *points* (also called „railroad switches“) and establishes safe routes for the trains to pass a railway junction or an entire rail yard. The *rail yard* is a complex series of railroad tracks and related points, track circuits, coloured signals, etc., e.g. for separation of trains to individual cars and their subsequent loading, unloading, and combination according to their cargoes and destinations.

interface (MMI), i.e. simulates operators, enters specific control commands, and monitors feedback information.

The SUT is represented by a *computer based control system* (CBCS). It runs the *control software*, interacts with operators (and testers, see above) by means of the MMI, and monitors and controls *external systems* of rail yards³ via an *external systems interface* (e.g. monitors sensors of the rail yards’ track circuits and controls actuators of their points³).

Each rail yard has its own instance of the testing environment with specific external system simulators derived from particular external systems. To implement a system for distribution and execution of the tests, which are represented by test scripts, over various instances of the testing environments, it has been proposed in [DMM⁺08] to use a service-oriented architecture (see Chapter 4). The system consists of a test manager and a set of testing environments. Available testing environments are registered by a broker and provided to the test manager at its request. Then, the test manager is able to receive a test script and execute it in an instance of a specific testing environment.

In the remaining sections of this chapter, we will focus on a *description of the testing environment* as the service-oriented architecture and an underlying component-based system. The environment will be described by means of the component model from Chapter 5 and the behavioural modelling of services from Chapter 6. We will prove that the proposed approach can be used in the practice. Formally described behaviour of services and components will allow us to make simulations of the behaviour, to detect deadlocks, and to check strong and weak open bisimulation equivalences between behaviours of different services and components. This will be useful, especially to check the *test scripts*, which are processed by the *tester*, and to control the tester’s behaviour and communication with other parts of the environment and with SUT. The wrong behaviour or the erroneous communication can cause the tests to fail and, moreover, may block future requests to the testing environment (for details, see Section 9.6.2).

9.2 Service Identification

From the description of the testing environment and the system’s architecture, the following tasks can be identified as invocations of services: „Submit Test“, „Execute Test“, „Log Results“, „Read Log“, „Publish Environment“, and „Find Environment“. The tasks can be implemented by the following business (entity) services, as it is described in Figure 9.2⁴: `TestManager`, `TestEnvironment`, `TestEnvironmentBroker`, and `TestLogger`.

⁴ The UML component diagram describes identified services by means of the notation from Section 6.3.1, which has been published in [RW08].

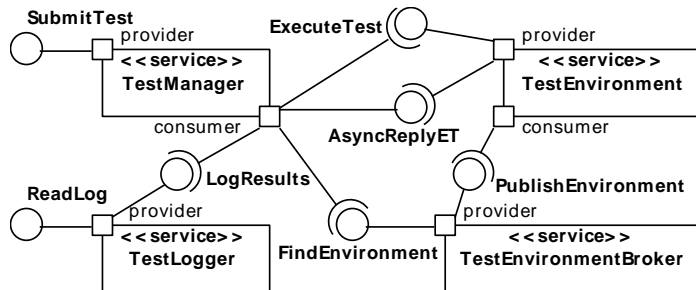


Fig. 9.2. An overview of identified services of the testing environment and their interconnections.

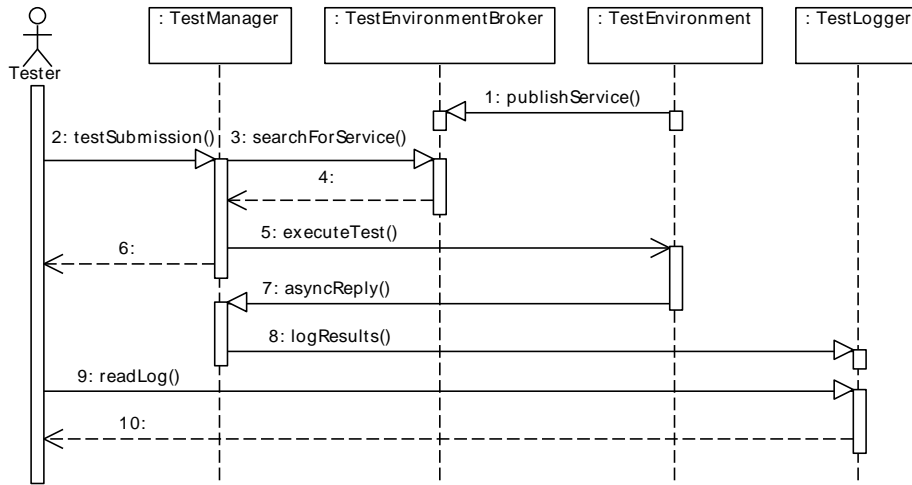


Fig. 9.3. The choreography of services in the testing environment.

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

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

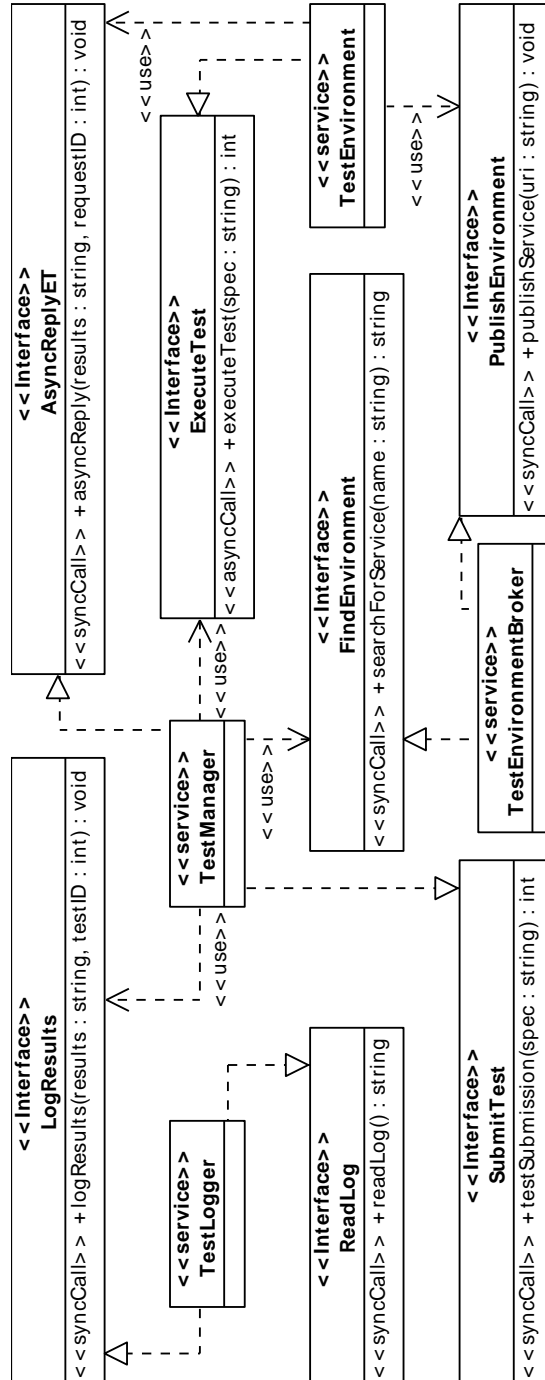


Fig. 9.4. Services of the testing environment as UML classes.

Figure 9.3 shows a choreography of the services in the testing environment as an UML sequence diagram⁵. Detailed description of the services as classes and their interfaces with relevant stereotypes is described in the UML class diagram⁵ in Figure 9.4. Service `TestEnvironment` is invoked asynchronously via `ExecuteTest` (see Section 6.3.1), i.e. a reply corresponding to the request will be returned later via the service’s interface `AsyncReplyET`.

9.3 Component-Based System

Railway interlocking control systems are safety-critical systems and can be described as component-based systems [BS93]. A testing environment of such systems has to interact with the systems’ components, as it is described in Section 9.1. For that reason, a part of the testing environment, which is directly connected to a system under testing (via the external systems simulators, see Figure 9.1), has character of a component neighbouring to the system under testing and therefore, it can be described by means of the component model from Chapter 5.

Figure 9.5 shows composite component `testEnvironment`, which represents service `TestEnvironment` from Section 9.2, by means of the notation from Section 5.1.2. The composite component consists of components `controller`, `environment`, `test`, and `output`.

Component `testEnvironment` receives a test script represented by a fresh copy (a clone) of a specific component via provided interface `executeTest`. The test script is processed by component `controller`, which attaches the new component as a subcomponent `test` of component `testEnvironment` by means of its control interface `teAttachP`. The controller also binds interfaces⁶ `tInteract` and `tResult` of component `test` to interface `eInteract` of component `environment` and interface `oResult` of component `output`, respectively. Then, component `test` is activated via interface `startTestP` and executed with a new identifier via interface `executeWithID`. The identifier is also returned by component `testEnvironment` as a reply of the test script’s submission.

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

After component `controller` is notified about the finished test script, it is able to receive and execute another test script, i.e. to attach a new component

⁵ The notation used in the UML sequence diagram and in the UML class diagram is described in Section 6.3.2.

⁶ According to Section 5.1.1, control interfaces can not be dynamically bound. The control interfaces of subcomponent `test` are bound as a part of its nesting into the component `testEnvironment`, which is permitted (see Section 5.2.4).

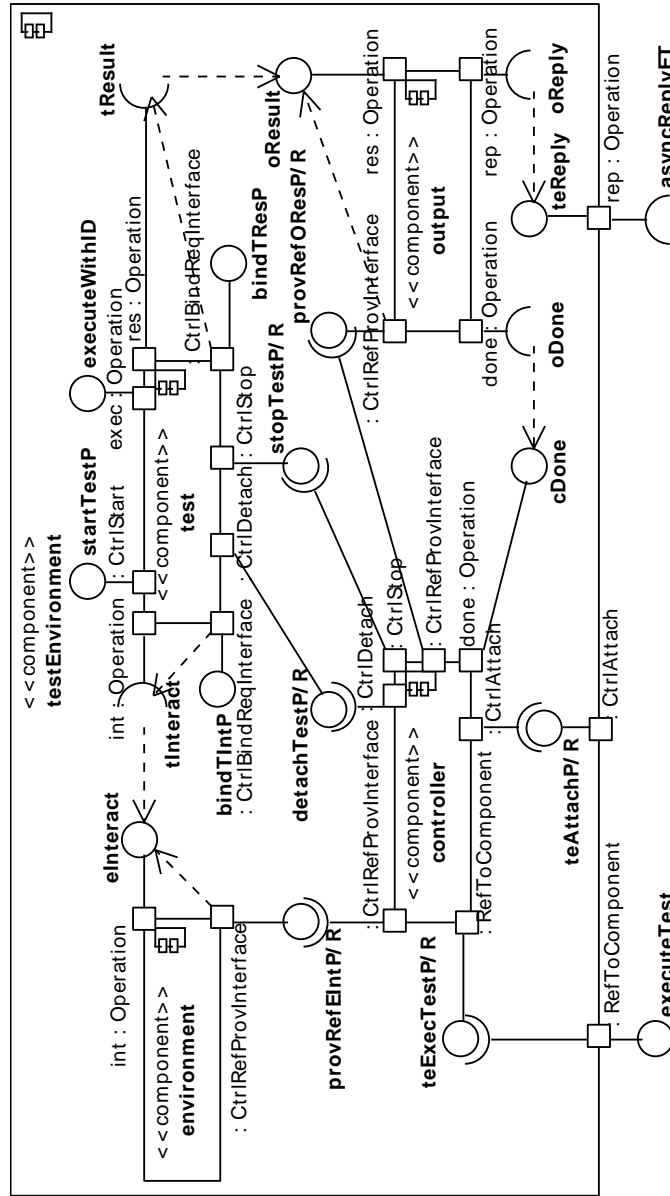


Fig. 9.5. Structure of composite component TestEnvironment with attached component test.

in the place of component test. Before that, component test with the old script is stopped via interface `stopTestP` and detached via its control interface `detachTestP`⁷.

9.4 Formal Description of the Service-Oriented Architecture

In this section, we describe behaviour of the services in the testing environment. Behaviour of services `TestManager`, `TestEnvironmentBroker`, `TestEnvironment`, and `TestLogger` can be described by means of π -calculus process abstractions TM , TEB , TE , and TL , respectively, according to Chapter 6. These process abstractions use names st , pe , fe , et , ar , lr , and rl as representations of the services’ interfaces `SubmitTest`, `PublishEnvironment`, `FindEnvironment`, `ExecuteTest`, `AsyncReplyET`, `LogResults`, and `ReadLog`, respectively.

According to Section 6.1, process abstraction TM describing behaviour of service `TestManager` is defined as follows:

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

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

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

⁷ In the diagram in Figure 9.5, only these two interfaces of `test` are connected with `controller`, because the rest of the test’s interfaces are used only during its nesting and their connections do not exist outside of `controller` component (see Section 9.5).

9.4 Formal Description of the Service-Oriented Architecture 103

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

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

where pe and fe are names representing the service’s interfaces **PublishEnvironment** and **FindEnvironment**, respectively, and subsequently processed by the constant applications of TEB_{pub} and TEB_{find} . Process abstraction TEB represents behaviour of a service broker, as it has been described in Section 6.1.1 (see the process abstraction *Broker*).

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

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

where et , ar , and pe are names representing the service’s interfaces **ExecuteTest**, **AsyncReplyET**, and **PublishEnvironment**, respectively. Initialisation of the service is described as process abstraction TE_{init} , which sends the service’s interfaces represented by names et and ar via name pe (i.e. publishes the corresponding interfaces via interface **PublishEnvironment**). After the initialisation, names et and ar are processed by pseudo-application $TE_{impl}\langle et, ar \rangle$, which describes behaviour of a component-based system implementing the service (see Section 6.2) and will be defined in Section 9.5.

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

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

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

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

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

Behaviour of the whole system of the interconnected services can be described as process abstraction $System$, which provides names st and rl representing interfaces `SubmitTest` and `ReadLog`, respectively, and which is defined as follows⁸:

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

9.5 Formal Description of the Component-Based System

The previous section has described behaviour of the testing environment as the service-oriented architecture from Section 9.2. All processes, which represent behavioural descriptions of individual services, have been described completely, except for process abstraction TE of service `TestEnvironment`. This service is implemented as a component-based system according to Section 6.2 with behaviour described by pseudo-application $TE_{comp}\langle s_0, s_1, c, ar^s, et^g \rangle$ (see Section 9.4).

In this section, we describe behaviour of primitive components `controller`, `environment`, `test`, and `output`, as process abstractions Ctr , Env , $Test$, and Out , respectively, and their parent composite component `testEnvironment`, as process abstraction TE_{comp} from the previous section.

Core behaviour of primitive components `output` and `controller` (i.e. the behaviour without default control actions, see Section 5.2.5) can be defined⁹ as process abstractions Out_{core} and Ctr_{core} , respectively, as follows:

⁸ We assume that connections of the services are static, without service brokers, except for service broker `TestEnvironmentBroker`, which has been described by process abstraction TEB (it can be compared with process abstraction $System$ from Section 6.3.3).

⁹ For notation of names in π -calculus processes describing components, see Section 5.2.1. The components' provided or required interfaces are represented by names $p\dots$ or $r\dots$, respectively, without the last character of the names ($\dots P/R$, see Figure 9.5).

9.5 Formal Description of the Component-Based System 105

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

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

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

106 9 Case Study

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

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

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

According to Section 5.2.5, behaviour of components *output*, *environment*, and *test* including their control parts can be defined as process abstractions Out , Env , and $Test$, respectively, as follows:

$$\begin{aligned}
 Out &\stackrel{def}{=} (s_0, s_1, c, p_{oResult}^g, p_{oDone}^s, p_{oReply}^s).(p_{oResult}, r_{oDone}, r_{oReply}) \\
 &\quad (Ctrl_{Ifs}\langle p_{oResult}, p_{oResult}^g \rangle \mid Ctrl_{Ifs}\langle r_{oDone}, p_{oDone}^s \rangle \\
 &\quad \mid Ctrl_{Ifs}\langle r_{oReply}, p_{oReply}^s \rangle \mid Ctrl_{clone}[c] \\
 &\quad \mid Out_{core}\langle p_{oResult}, r_{oDone}, r_{oReply} \rangle) \\
 Env &\stackrel{def}{=} (s_0, s_1, c, p_{eInteract}^g).(p_{eInteract}) \\
 &\quad (Ctrl_{Ifs}\langle p_{eInteract}, p_{eInteract}^g \rangle \mid Ctrl_{clone}[c] \mid Env_{core}\langle p_{eInteract} \rangle) \\
 Test &\stackrel{def}{=} (s_0, s_1, c, p_{executeWithID}^g, p_{tInteract}^s, p_{tResult}^s). \\
 &\quad (p_{executeWithID}, r_{tInteract}, r_{tResult}) \\
 &\quad (Ctrl_{Ifs}\langle p_{executeWithID}, p_{executeWithID}^g \rangle \\
 &\quad \mid Ctrl_{Ifs}\langle r_{tInteract}, p_{tInteract}^s \rangle \mid Ctrl_{Ifs}\langle r_{tResult}, p_{tResult}^s \rangle \\
 &\quad \mid Ctrl_{clone}[c] \mid Test_{core}\langle p_{executeWithID}, r_{tInteract}, r_{tResult} \rangle)
 \end{aligned}$$

where names s_0, s_1, c, p_{\dots}^g and p_{\dots}^s have been described in Section 5.2.1, process abstraction $Ctrl_{Ifs}$ has been defined in Section 5.2.2, and process constant $Ctrl_{clone}$ has been defined in Section 5.2.4.

Behaviour of component *controller* has to be defined differently from the others, because it uses required control interfaces represented by names $r_{teAttach}$, $r_{detachTest}$, $r_{stopTest}$, $r_{provRefEInt}$, and $r_{provRefORes}$, which can not be referenced (contrary to functional interfaces, see Section 5.1.1 and the example from Section 5.3). The behaviour is defined by means of process abstraction $Ctrl$ as follows:

9.5 Formal Description of the Component-Based System 107

$$\begin{aligned}
 Ctr \stackrel{def}{=} & (s_0, s_1, c, p_{cDone}^g, p_{teExecTest}^g, \\
 & r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes}). \\
 & (p_{cDone}, p_{teExecTest}) \\
 & (Ctrl_{Ifs} \langle p_{cDone}, p_{cDone}^g \rangle \mid Ctrl_{Ifs} \langle p_{teExecTest}, p_{teExecTest}^g \rangle \\
 & \mid Ctrl_{clone} [c] \mid Ctr_{core} \langle p_{cDone}, p_{teExecTest}, \\
 & r_{teAttach}, r_{detachTest}, r_{stopTest}, r_{provRefEInt}, r_{provRefORes} \rangle)
 \end{aligned}$$

Finally, and according to Section 5.2.5, behaviour of composite component `testEnvironment`, which represents the whole component-based system implementing the core of service `TestEnvironment` (see pseudo-application $TE_{comp} \langle s_0, s_1, c, et^g, ar^s \rangle$ in Section 9.4), can be described as process abstraction TE_{comp} as follows:

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

$$Test_{plug} \stackrel{def}{=} (r_{detachTest}, r_{stopTest}).(r_{detachTest} \mid r_{stopTest})$$

where process abstractions $Ctrl_{EI}$ and $Ctrl_{SS}$ have been defined in Section 5.2.2 and Section 5.2.3, respectively.

Process abstraction TE'_{comp} , which is applied in process abstraction TE_{comp} , creates concurrent processes given by pseudo-applications of $Ctrl$, Out , and Env and sends their names s_0 and s_1 via name $p_{teAttach}$, i.e. attaches components **controller**, **output**, and **environment**, respectively, as subcomponents of component **testEnvironment**. It also interconnects names representing required and provided control interfaces of the components by means of three constant applications of process constant $Wire$ (see Section 5.2.2). Concurrently with the previous step, TE'_{comp} applies process abstraction $Test_{plug}$ and binds name $p_{teExecTest}$ of the pseudo-application of $Ctrl$ to name $r_{teExecTest}$ of the pseudo-application of TE_{comp} , name p_{cDone} of $Ctrl$ to name r_{cDone} of Out , and name $p_{teReply}$ of TE_{comp} to name $r_{teReply}$ of Out . The pseudo-application of process abstraction $Test_{plug}$ handles requests initiated by the pseudo-application of $Ctrl$ and received by names $r_{stopTest}$ and $r_{detachTest}$ to stop and to detach a process representing behaviour of a previous but non-existent component with a test script (e.g. a non-existent predecessor of component **test**).

9.6 System Properties and Their Verification

The behaviour formally described in the previous sections can be used for verification and model checking by means of the tools from Section 8.2. The utilisation is demonstrated by examples of interactive simulation in Section 9.6.1, finding deadlocks in Section 9.6.2, bisimulation checking in Section 9.6.3, and model checking in Section 9.6.4.

The examples utilise *The Mobility Workbench* (MWB) and *Another/Advanced Bisimulation Checker* (ABC), which have been described in Section 8.2.1 and Section 8.2.2, respectively. Complete transcription of process abstraction $System$ from Section 9.4 and the applied process abstractions and constants as agents of MWB and ABC can be found in Appendix A. Unguarded constant applications of recursive process constants are prefixed by unobservable prefix τ (see Section 8.2.1).

9.6.1 Simulation

To simulate behaviour of agent **System**, i.e. behaviour of the system from the case study (see process abstraction $System$ in Section 9.4 and the agent in Appendix A), we need to submit a sample test to the system, wait for its processing and finally, receive its results. Therefore, agent **Tester** is defined as follows:

```

agent Tester =
  (^s0, s1, pgexecuteWithID, pstInteract, pstResult, r1, st)
  (
    Test(s0, s1, pgexecuteWithID, pstInteract, pstResult)
    | System(st, r1) | (^ts, ret, r, p) 'st<ts, ret> .
    'ts<s0, s1, r, p> . 'r<pstInteract, pstResult> .
    'p<pgexecuteWithID> . ret(id1) . (^r2) 'r1<r2> .
    r2(res, id2) . 0 )

```

Agent **Tester** is a composition of the applications of agents **Test** and **System**, and an auxiliary π -calculus process (after the last composition operator). The auxiliary process submits all names of the application of agent **Test** (i.e. names $s0$, $s1$, $pgexecuteWithID$, $pstInteract$ and $pstResult$) indirectly via name st to the application of agent **System** and receives name $id1$ as a reply via name ret . Then, it waits for results of a test performed by the application of agent **Test**, which can be received via name $r1$ of the application of agent **System**.

Behaviour of agent **Tester** can be interactively simulated in MWB by means of command „**step Tester**“¹⁰.

9.6.2 Deadlocks

A *deadlock* occurs in a π -calculus process iff the process can not perform any reduction step, i.e. the process is not responding to any action on its free names (see Definitions 11 and 12 in Section 2.3).

To permit concurrent processing of multiple requests, process abstractions and constants TM_{st} , TM_{ar} , TEB_{pub} , TEB_{find} , TL_{lr} , TL_{rl} , Out'_{core} , and Env'_{core} , from Sections 9.4 and 9.5, and $SetIf$, $Ctrl_{lfs}$, and $Attach$, from Sections 5.2.2 and 5.2.3, use unguarded or weakly guarded recursions (i.e. guarded by unobservable prefix τ). These processes, as separate units, do not come to deadlocks, because each of them can always perform at least one reduction step (namely, reduction step R-TAU from Definition 11)¹¹.

Agents representing the processes from the case study (see Appendix A) have been checked for deadlocks, by means of command „**deadlocks**“ in MWB. In some cases, the deadlock-checking can not be finished due to the unguarded or weakly guarded recursions (see the previous paragraph). However, the *deadlocks have been found* in agents **TestCore**, **TestPlug**, **Wire**, **Dist**, **TE2comp**, and **TEimpl**.

Agents **TestCore**, **TestPlug**, **Wire**, and **Dist** have deadlocks in process 0, which is reachable by 1, 4, 2, and 1 commitments, respectively. These

¹⁰ However, the simulation is not transparent but demanding for an user because of large amount of possible internal (silent) actions.

¹¹ Nevertheless, these processes can come to a *live-lock* in their mutual co-operation. In such a case, the processes will communicate only between themselves and will periodically change , but as a whole system, they will not be responding to any external actions on their free names.

deadlocks are desired, since the agents represent process abstractions $Test_{core}$ and $test_{plug}$ (see Section 9.5) and process constants $Wire$ and $Dist$ (see Section 5.2.2 and Section 5.2.3, respectively), which describe finite behaviour and can be reduced to process 0 by input, output, and τ actions on their free names.

Process abstraction $Test_{core}$ describes behaviour of a core functionality of component `test`, which implements a test script. The behaviour is finished after the test script is performed, so $Test_{core}$ is reduced to process 0. Analogously, process abstraction $test_{plug}$, which describes processing of first requests to stop and to detach a non-existent component before it can be replaced by a real component implementing a specific test script (e.g. component `test`), is performed only once and reduced to process 0. Process constants $Wire$ and $Dist$ describe behaviour of a connector of two interfaces and distribution of a start/stop request from a composite component among its subcomponents, respectively. Although they contain recursions and their behaviour can be infinite, they can be terminated instantly (e.g. when the connector is removed or the request has been already submitted to all of the subcomponents). In such case, process constants $Wire$ or $Dist$ can be reduced to process 0 (by means of an input action on name d or an output action on name r , respectively; see Sections 5.2.2 and 5.2.3).

Agents `TE2comp` and `TEimpl` have deadlocks in processes that are reachable by 22 and 31 commitments, respectively. The deadlocks are related to the ability of process abstraction TE_{comp} , which describes behaviour of composite component `testEnvironment`, and of process abstraction TE , which describes behaviour of service `TestEnvironment`, to receive and to execute a test script. During the execution, behaviour of the component and the service is controlled by the test script (the component’s subcomponent `controller` is waiting for an input on its interface `cDone`, see Section 9.3). If the test script is incompatible with its environment and can not be finished, the component and the service come to a deadlock.

In our approach, the deadlock-checking can be utilised to detect erroneous behaviour of individual services and components.

9.6.3 Bisimulation Checking

The ABC allows to check strong and weak open bisimulation equivalences by means of commands „`eq`“ and „`weq`“. Moreover, in a case of two agents that have the same free names, the bisimulation equivalences can be checked also by means of commands „`eqd`“ and „`weqd`“, which suppose the free names of the first agent are distinct from the free names of the second agent. For details, see Definition 16 in Section 2.3.2.

To demonstrate bisimulation checking in our case study, we check the equivalences of process $Test_{core}$ and its possible replacements. The process describes core behaviour of component `test` representing a test script (see Section 9.5). The bisimulation checking of behaviour of the original test script,

9.6 System Properties and Their Verification 111

which is supposed to be correct, and behaviour of its replacements, which may be wrong, can prevent the deadlock in agents `TE2comp` and `TEimpl`, as it has been described in Section 9.6.2.

In addition to agent `TestCore`, we define two agents with the same free names. The following definitions include original agent `TestCore` and new agents `TestCoreEquiv` and `TestCoreNonequiv` (see also Appendix A.2):

```
agent TestCore =
  (\pexecuteWithID,rtInteract,rtResult)
  pexecuteWithID(id) . (^ret) 'rtInteract<ret> .
  ret(val) . 'rtResult<val,id> . 0

agent TestCoreEquiv =
  (\pexecuteWithID,rtInteract,rtResult)
  pexecuteWithID(id) . (^comm) ( (^ret)
  'rtInteract<ret> . ret(val) . 'comm<val> . 0 |
  comm(res) . 'rtResult<res,id> . 0 )

agent TestCoreNonequiv =
  (\pexecuteWithID,rtInteract,rtResult)
  pexecuteWithID(id) . (^ret) 'rtInteract<ret> .
  ret(val) . (^resid) 'rtResult<val,resid> . 0
```

Agents `TestCore` and `TestCoreEquiv` are not strongly open bisimilar, because agent `TestCoreEquiv` can perform an internal communication via name `comm` (according to rule L-COMM in Definition 13 from Section 2.3.1), that can not be performed by agent `TestCore`. However, these agents are weakly open bisimilar and according to ABC, a core relation¹² of their bisimulation contains 12 members.

The agents `TestCore` and `TestCoreNonquiv` are neither strongly open bisimilar nor weakly open bisimilar. The problem is at the end of processing, when agent `TestCore` sends via name `rtResult` name `id`, which has been previously received via name `pexecuteWithID`, while agent `TestCoreNonquiv` creates and sends a fresh name `resid`, which differs from the original name `id`. The replacement of agent `TestCore`, which describes behaviour of component `test`, by agent `TestCoreNonequiv` leads to a deadlock (see the context of component `test` in Section 9.3).

¹² The core relation of bisimulation is a ternary relation between an agent, a set of distinctions, and an other agent, such that an union of its symmetric closure and the identity relation is a bisimulation [Bri05].

9.6.4 Model Checking

Model checking is possible by means of the MWB, which uses π - μ -calculus [Dam94], an extension of the μ -calculus¹³, as a property specification language.

In MWB, we can check safety and liveness properties by means of μ and ν operators, respectively, as well as simply check the existence of specific reduction steps by means of modal operators \diamond and \square . The following command verifies the ability of agent `System` to perform input actions on its free names `st` and `rl`:

```
check System(st,rl) <st>TT & <rl>TT
```

Agent `System` describes behaviour of the system from our case study (see process abstraction `System` in Section 9.4 and the agent in Appendix A). The complete description of syntax and semantics of the π - μ -calculus in MWB can be found in [Vic95].

9.7 Evaluation and Conclusion

In the previous sections, we have demonstrated the application of service-oriented architectures, the component model, and the behavioural description of component-based systems and services, as it has been proposed in Chapters 5 and 6. The case study of a service-oriented architecture for functional testing of complex safety-critical systems has been introduced in Section 9.1 and modelled as the service-oriented architecture in Section 9.2 and as the component-based system in Section 9.3. In Sections 9.4 and 9.5, we have formally described behaviour of services of the architecture and components of the system, respectively. Finally, in Section 9.6, the behaviour has been simulated, checked for deadlocks, strong and weak open bisimulations, and its verification has been outlined, by means of the tools from Section 8.2.

Through the case study, we have successfully validated the proposed modeling approaches. To evaluate the results, we will compare our approach and important features of the case study solution with the related approaches from Chapters 3 and 4. The related approaches 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 mentioned in Section 4.1.1 (e.g. transformations of BPEL to Petri nets [HSS05] or to π -calculus processes [LM07, WDW07]);

¹³ The (modal) μ -calculus is a temporal logic with a least fix-point operator μ and a greatest fix-point operator ν . It is used to specify properties of concurrent systems represented as labelled transition systems (see Section 2.1).

2. *formal approaches to modelling of component-based systems*, such as component models and architecture description languages mentioned in Sections 3.3 and 3.4, which are usually focused only on CBSs without consideration of SOA at the higher level of abstraction (e.g. Wright [AG96], Tracta [Gia99], behaviour protocols of SOFA [Viš02], formal descriptions of Fractive [Bar05], and, partially, SOFA 2.0 [BHP06]).

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 (see Section 3.3.4) and the Reo coordination language [DA04].

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 [Arb04, DA04] 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 (see process abstraction *System* in Section 9.4). 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, as it has been described in Section 9.6.

9.7.1 Important Merits

In comparison to the approaches mentioned above, 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 (see Section 3.1).
- 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 (see Section 5.1.1).
- Behaviour of services and components is described in the π -calculus, which has a *native support for reconfiguration and mobility*. The π -calculus is a suitable formal basis for behavioural description of systems with mobile architectures (see Section 3.1).

- 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 (see Section 8.2 and Section 9.6).
- 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 (see Section 4.3). 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.

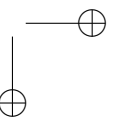
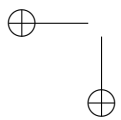
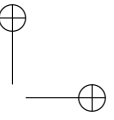
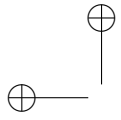
9.7.2 Possible Drawbacks

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 (see Section 8.2.1 and Section 9.6).
- The representation of system models uses the *specific and informal UML-like notation* (see Section 5.1.2), instead of a formally defined UML 2 profile (see Section 3.4.2).
- 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.
- The approach, which is presented in this book, 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 (see Section 9.6.2).

Part IV

Conclusion



10

Summary

This book provides a review of the state of the art of modelling of component-based systems and introduces a novel component model for description of mobile architectures. In comparison with the current state of the art, i.e. the related approaches presented in Section 3.3, the proposed component model has advantages in support of fully dynamic architectures including mobility of their entities, 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. These features are related to the problem factors F1–F5, which have been described in Section 1.1.

The book meets the objectives set out in Sections 1.1.1–1.1.5. 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 as π -calculus processes. Moreover, the formal description has included the behavioural description of service-oriented architectures. 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 also outlined the integration of our approach into a development process and 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

118 10 Summary

on the levels of the service-oriented architecture and the component-based system.

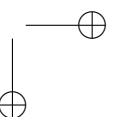
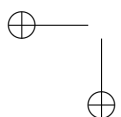
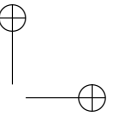
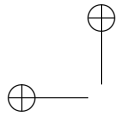
Finally, the formally described services and components have been simulated, checked for deadlocks, strong and weak open bisimulation equivalence, and verification of their properties has been outlined.

11

Future Research Directions

The research presented in this book has introduced the component model with formal basis and support of mobile architectures. Its potential improvements can result from missing features, which are supported by the related approaches (see Chapter 3), as well as from elimination of the possible drawbacks (see Section 9.7.2). There are several lines of research arising from this work which should be pursued:

- An UML 2 *profile* (see Section 3.4.2) for modelling of component-based systems should be developed based on the component model’s metamodel. It may replace the informal UML-like notation from Section 5.1.2.
- The tool for modelling of component-based systems, which has been introduced in Section 8.1.2, should be *updated to reflect the current metamodel* of the component model. In further work, the tool may be extended to provide *automatic generation of behavioural description* of a system according to behavioural description of its primitive components and a model of its structure (see Section 7.2). The extension will improve integration of the tool into software development processes and reduce the qualification requirements for developers (see the third item in Section 9.7.2).
- To support an implementation phase of the software development process (see Chapter 7), the component model should be *integrated into various component-based environments*, e.g. CORBA components, Java EE components (JavaBeans) or Microsoft component technologies (COM model). Moreover, an *implementation framework* should be provided.
- The behavioural description of services and components in π -calculus should be modified in order to eliminate unguarded or weakly guarded applications (infinite recursions), which can cause decidability issues (see Sections 8.2.1 and 9.6). This may include a modification of the behavioural description to use a specific variant of the π -calculus or a specific language based on the π -calculus. However, such modification will eliminate ability to utilise existing tools for model checking of standard π -calculus processes and formal verification of their properties (see Section 9.7.1).



References

- ACD⁺03. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for Web Services, version 1.1. Technical report, IBM, May 2003.
- AG96. 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.
- AGM04. 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.
- AK06. Atif Aziz and Jan-Klaas Kollhof. JSON-RPC 1.1 specification. Working draft, August 2006.
- AM02. Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *17th IEEE International Conference on Automated Software Engineering (ASE’02)*, pages 271–274, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- Ams05. Jim Amsden. Business services modeling: Integrating WebSphere business modeler and rational software modeler, December 2005. IBM developerWorks.
- AN05. 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.
- Arb04. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.
- Arc06. ArchWare project. <http://www.arch-ware.org/>, November 2006.
- Ars04. Ali Arsanjani. Service-oriented modeling and architecture: How to identify, specify, and realize services for your SOA, November 2004. IBM developerWorks.

122 References

- Bar05. Tomás Barros. *Formal specification and verification of distributed component systems*. PhD thesis, Université de Nice – INRIA Sophia Antipolis, November 2005.
- BBC⁺06. Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.
- BCK03. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley Professional, second edition, 2003.
- BCM03. Françoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242. Springer-Verlag, 2003.
- BCMR06. Tomás Barros, Antonio Cansado, Eric Madelaine, and Marcela Rivera. Model-checking distributed components: The Vercors platform. In Frank S. de Boer and Vladimír Mencl, editors, *Preliminary Proceedings of the Third International Workshop on Formal Aspects of Component Software FACS’06*, number 344 in UNU-IIST Reports, P.O.Box 3058, Macau, September 2006.
- BCS02. 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.
- BCS04. Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium, February 2004.
- BH95. Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.
- BH06. Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods . . . ten years later. *Computer*, 39(1):40–48, 2006.
- BHP06. 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.
- BMO⁺05. 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.
- BN07. Sébastien Briaïs and Uwe Nestmann. Open bisimulation, revisited. *Theoretical Computer Science*, 386(3):236–271, November 2007.
- Bri05. Sébastien Briaïs. *The ABC User’s Guide*, May 2005.
- BS93. Jonathan P. Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.
- BSM⁺03. Frank Budinsky, Dave Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, August 2003.

- CCC⁺07. Philippe Collet, Thierry Coupaye, Hervé Chang, Lionel Seinturier, and Guillaume Dufrière. Components and services: A marriage of reason. Technical Report ISRN I3S/RR-2007-17-FR, Project RAINBOW, CNRS, May 2007.
- CCL06. 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.
- Cha07. David Chappell. Introducing SCA. White paper, Chappell & Associates, 2007.
- CHvRR04. Luc Clement, Andrew Hatley, Claus von Riegen, and Tony Rogers. UDDI version 3.0.2. Uddi spec technical committee draft, OASIS Open, October 2004.
- CKO92. Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.
- CMRW07. Roberto Chinnici, Jean-Jacques Moreau, Arthur Rymann, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. W3c recommendation, W3C, June 2007.
- ČVZ06. Ivana Černá, Pavlína Vařeková, and Barbora Zimmerová. Component-interaction automata modelling language. Technical Report FIMURS-2006-08, Faculty of Informatics, Masaryk University, October 2006.
- CW96. Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
- DA04. 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.
- dAH01. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.
- Dam94. Mads Dam. Model checking mobile processes (full version). SICS Research Report R94:01, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden, 1994.
- DMM⁺08. Renato Donini, Stefano Marrone, Nicola Mazzocca, Antonio Orazzo, Domenico Papa, and Salvatore Venticinque. Testing complex safety-critical systems in SOA context. In *CISIS*, pages 87–93, Los Alamitos, CA, USA, December 2008. IEEE Computer Society.
- Ecl07a. The Eclipse Foundation. Eclipse graphical modeling framework (GMF). <http://www.eclipse.org/gmf/>, September 2007.
- Ecl07b. The Eclipse Foundation. Eclipse modeling framework project (EMF). <http://www.eclipse.org/modeling/emf/>, September 2007.
- Erl05. Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, August 2005.

124 References

- Eva94. Andy S. Evans. Specifying & verifying concurrent systems using Z. In *FME'94 Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, 1994.
- Fie00. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
- Gal09. Ivan Gal. A tool for modelling of component-based systems. Master’s thesis, Brno University of Technology, Faculty of Information Technology, Department of Information Systems, June 2009.
- Gdh07. Joe Gregorio and Bill de hOra. The Atom publishing protocol. IETF RFC 5023, October 2007.
- GHM⁺07. Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP version 1.2 part 1: Messaging framework. W3c recommendation, W3C, April 2007.
- Gia99. 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.
- GMW00. 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.
- Hal90. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.
- HHS06. Denis Hatebur, Maritta Heisel, and Jeanine Souquières. A method for component-based software and system development. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 72–80. IEEE Computer Society, 2006.
- HP06. 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.
- HSS05. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets. In *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, Nancy, France, September 2005. Springer-Verlag.
- IB07. Srikanth Inaganti and Gopala Krishna Behara. Service identification: BPM and SOA handshake. *BP Trends*, March 2007.
- IEE00. 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.
- ISO02. Information technology – Z formal specification notation – syntax, type system and semantics. International Standard ISO/IEC 13568:2002, July 2002.
- Kru95. Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

- KŽ00. Jaroslav Král and Michal Žemlička. Autonomous components. In *SOFSEM 2000: Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*, pages 375–383. Springer, 2000.
- KŽ03. Jaroslav Král and Michal Žemlička. Software confederations and alliances. In *CAiSE Short Paper Proceedings*, volume 74 of *CEUR Workshop Proceedings*, pages 229–232. CEUR-WS.org, 2003.
- Lin00. Huimin Lin. Computing bisimulations for finite-control π -calculus. *Journal of Computer Science and Technology*, 15(1):1–9, 2000.
- LK06. Beate List and Birgit Korherr. An evaluation of conceptual business process modelling languages. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1532–1539, New York, NY, USA, April 2006. ACM.
- LM07. 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.
- LW05. 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.
- LW06. 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.
- MB05. Vladimír Mencl and Tomáš Bureš. Microcomponent-based component controllers: A foundation for component aspects. In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 729–737, Taipei, Taiwan, December 2005. IEEE Computer Society Press.
- MDEK95. 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.
- Men98. Vladimír Mencl. Component definition language. Master’s thesis, Charles University, Prague, May 1998.
- Mio06. Matteo Mio. PiET – pi calculus equivalences tester. <http://piet.sourceforge.net/>, September 2006.
- MMP⁺95. Richard J. Mayer, Christopher P. Menzel, Michael K. Painter, Paula S. deWitte, Thomas Blinn, and Benjamin Perakath. Information integration for concurrent engineering (IICE) IDEF3 process description capture method report. Technical report, Knowledge Based systems Incorporated (KBSI), September 1995.
- MPW92. 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.
- MRRR02. 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.

126 References

- MT00. 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.
- ODvdAtH06. Chun Ouyang, Marlon Dumas, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede. From business process models to process-oriented software systems: The BPMN to BPEL way. Technical report, October 2006.
- OMG98. 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.
- OMG05a. 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.
- OMG05b. 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.
- OMG06a. Meta object facility (MOF) core specification, version 2.0. Document formal/06-01-01, The Object Management Group, January 2006.
- OMG06b. UML profile and metamodel for services (UPMS), request for proposal. Document soa/2006-09-09, The Object Management Group, September 2006.
- OMG07a. UML infrastructure, version 2.1.2. Document formal/2007-11-04, The Object Management Group, November 2007.
- OMG07b. UML superstructure, version 2.1.2. Document formal/2007-11-02, The Object Management Group, November 2007.
- OMG08a. Business process definition metamodel (BPDM), version 1.0. Document formal/2008-11-05, The Object Management Group, November 2008.
- OMG08b. Business process modeling notation (BPMN), version 1.1. Document formal/2008-01-17, The Object Management Group, January 2008.
- Ope07. Open SOA Collaboration. SCA XML schema files. <http://www.osoa.org/xmlns/sca/1.0/>, November 2007.
- Ope08. Open SOA Collaboration. Service Component Architecture specifications. <http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications>, July 2008.
- Oqu04. 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.
- Oqu05. Flavio Oquendo. UML 2.0 profile for ArchWare ADL. Technical Report D1.8, ArchWare Project IST-2001-32360, June 2005.
- OSO07a. SCA service component architecture: Assembly model specification. Technical Report SCA version 1.00, Open SOA Collaboration, March 2007.
- OSO07b. SCA policy framework. Technical Report SCA version 1.00, Open SOA Collaboration, March 2007.
- OSO07c. SCA service component architecture: ACID transaction policy in SCA. Technical Report SCA version 1.00, Open SOA Collaboration, December 2007.

- PBJ98. 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.
- PGG⁺06. Thomas Pfaeffle, Simeon M. Greene, Sumit Gupta, Bill Jones, Tim Julien, Gigi Lee, Mike Lehmann, Jon Maron, Kevin Minder, Bob Naugle, Eric Rajkovic, Ekkehard Rohwedder, Shih-Chang Chen, and Quan Wang. *Oracle Application Server Web Services Developer’s Guide, 10g (10.1.3.1.0)*. Oracle, September 2006.
- Ros98. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA, 1998.
- RSS07. RSS 2.0 specification. Technical report, RSS Advisory Board, October 2007.
- RW08. 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.
- Ryc08. Marek Rychlý. Behavioural modeling of services: from service-oriented architecture to component-based system. In *Software Engineering Techniques in Progress*, pages 13–27. Wroclaw University of Technology, October 2008.
- Ryc09. Marek Rychlý. A component model with support of mobile architectures and formal description. *e-Informatica Software Engineering Journal*, 3(1):9–25, October 2009.
- Sch00. August-Wilhelm Scheer. *Aris – Business Process Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.
- SW01. Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):664–676, November 2001.
- SW03. Davide Sangiorgi and David Walker. *The π -Calculus: A Theory of Mobile Processes*. Cambridge University Press, New Ed edition, October 2003.
- Szy02. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, second edition, November 2002.
- Ves93. Steve Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, February 1993.
- Vic94. Björn Victor. *A Verification Tool for the Polyadic π -Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.
- Vic95. Björn Victor. *The Mobility Workbench User’s Guide*, polyadic version 3.122 edition, October 1995.
- Viš02. Stanislav Višňovský. *Modeling software components using behavior protocols*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002.

128 References

- WDW07. 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.
- Win99. Dave Winer. XML-RPC specification. Technical report, UserLand Software, June 1999.

Acronyms

ABC	Another/Advanced Bisimulation Checker (a tool that checks open-equivalence in the π -calculus, see Section 8.2.2)
ADL	architecture description language (a language for describing software systems’ architectures, see Section 3.4)
ALTL	linear temporal logic of actions (an extension of linear temporal logic, see Section 3.3.2)
BP	business process
BPD	business process diagram (a representation of a business process model, see Section 4.1.1)
BPEL	Business Process Execution Language (a language for business process models, see Section 4.1.1)
BPEL4WS	Business Process Execution Language for Web Services (a language for business process models of Web Services, see Section 4.1.1)
BPM	business process model (a specification of business processes, see Section 4.1)
BPMN	Business Process Modelling Notation (a notation for describing business process models, see Section 4.1.1)
BSM	business services model (a mediator between business requirements and an implementation, see Section 4.1.2)
CBCS	computer based control system
CBD	component-based development (a software development methodology of component-based systems, see Section 3.2)
CBS	component-based system (a system that is composed of components, see Section 3.2)

130 References

- CBSE component-based software engineering
(a software development methodology of component-based systems, see Section 3.2)
- CCS Calculus of Communicating Systems
(a process calculus to model indivisible communications between exactly two participants)
- CDL Component Definition Language
(a specification language of software components in SOFA, see Section 3.3.3)
- CMOF Complete Meta Object Facility
(a part of Meta Object Facility that extends EMOF, see Section 5.1.1)
- COM Component Object Model
(an application interface for software components introduced by Microsoft)
- CORBA Common Object Requesting Broker Architecture
(a standard for software components defined by the Object Management Group)
- CSP Communicating Sequential Processes
(a formal language for describing patterns of interaction in concurrent systems, see Section 2.2)
- EMF Eclipse Modeling Framework
(an Eclipse framework for modelling and code generation of tools based on metamodels)
- EMOF Essential Meta Object Facility
(a part of Meta Object Facility with modelling elements for simple metamodels, see Section 5.1.1)
- FIFO first-in-first-out
(a data structure)
- FSP Finite State Processes
(a language/algebra for behavioural specification of components as finite LTSs, see Section 3.3.2)
- GMF Eclipse Graphical Modeling Framework
(an Eclipse framework for developing graphical editors of models based on metamodels in EMF)
- HTTP Hypertext Transfer Protocol
(an application-level protocol for distributed, collaborative, hypermedia information systems)
- IT information technology
- JSON JavaScript Object Notation
(a language-independent computer data interchange format)
- LTL linear temporal logic
(a modal temporal logic with modalities referring to time)

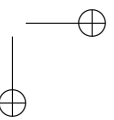
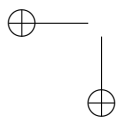
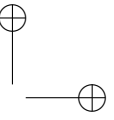
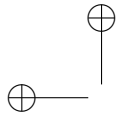
LTS	Labelled Transition System (a state transition system with labelled transitions, see Section 2.1)
MIME	Multipurpose Internet Mail Extensions (an internet standard for description of content of e-mail messages)
MMI	man machine interface
MOF	Meta Object Facility (a standard for model-driven engineering from the Object Management Group, see Section 5.1.1)
MWB	The Mobility Workbench (a model checker, bisimulation checker and verification tool for the π -calculus, see Section 8.2.1)
PiET	Pi-Calculus Equivalences Tester (a tool for checking of various types of equivalences of the π -calculus processes, see Section 8.2.3)
pLTS	Parametrised Labelled Transition System (a LTS with parametrised actions as labels and variables for states and a system, see Section 3.3.5)
pNet	Parametrised Synchronisation Network (a composition of pLTSs by parametrised sorts, global actions, and a transducer, see Section 3.3.5)
REST	Representational State Transfer (a style of software architecture for distributed hypermedia systems)
RPC	Remote Procedure Call (an inter-process communication technology)
RSS	Really Simple Syndication (a family of formats for syndication of web-content)
SCA	Service Component Architecture (an approach for design and implementation of SOA as CBSs, see Section 4.3.1)
SCDL	Service Component Definition Language (a XML-based language for description of compositions of SCA components, see Section 4.3.1)
SOA	service-oriented architecture (an architectural style for aligning business and IT architectures, see Chapter 4)
SOAP	Simple Object Access Protocol (a protocol for exchanging structured information between Web Services, see Section 4.2)
SOFA	SOFTware Appliances (a component model with support of dynamic architecture, see Section 3.3.3)
SUT	system under testing

132 References

- SWA software alliances
(software as networks of cooperative units formed temporarily during its runtime, see Chapter 1)
- SWC software confederation
(systems as networks of quite cooperative, permanently available services, see Chapter 1)
- TS transition system
(an abstract machine with a set of states and transitions between these states, see Section 2.1)
- UDDI Universal Description, Discovery and Integration
(a registry for publishing and discovering Web Services listings, see Section 4.2)
- UML Unified Modeling Language
(a standardised modelling language for software systems, see Section 3.4.2)
- URI Uniform Resource Identifier
(an identification, a location or a name, of a resource on the Internet)
- WSDL Web Services Description Language
(an XML-based language for description of Web Services, see Section 4.2)
- XMI XML Metadata Interchange
(a standard for exchanging metadata information in XML from the OMG, see Section 8.1.2)
- XML Extensible Markup Language
(a language for specification of structured documents and their processing)

Part V

Appendices



A

Process Descriptions from the Case-Study in MWB/ABC

In this appendix, we describe the π -calculus process abstractions and constants, which have been defined in Section 9.4 and Section 9.5 of the case study from Chapter 9. The following process abstractions and constants are adapted to The Mobility Workbench (MWB, see Section 8.2.1), but they are compatible also with Another/Advanced Bisimulation Checker (ABC, see Section 8.2.2). The agents are used in Section 9.6 for interactive simulation, finding deadlocks, bisimulation equivalences and model checking.

A.1 Control Parts of Components

The following agents are adapted from the processes from Section 5.2 and describe behaviour of control parts of components.

```

1  (** Wire **)
2  agent Wire = (\x,y,d) ( x(m) . 'y<m> . Wire(x,y,d) +
   d . 0 )
3  (** CtrlIfsR **)
4  agent CtrlIfsR = (\r,ps) (^d) ( d . 0 | SetIf(r,ps,d)
   )
5  agent SetIf = (\r,s,d) s(p) . ( 'd . Wire(r,p,d) |
   t.SetIf(r,s,d) )
6  (** CtrlIfsP **)
7  agent CtrlIfsP = (\p,pg) pg(r) . ( 'r<p> . 0 |
   t.CtrlIfsP(p,pg) )
8  (** CtrlEIR **)
9  agent CtrlEIR = (\re,pi) (^d) Wire(re,pi,d)
10 (** CtrlEIP **)
11 agent CtrlEIP = (\pe,ri) (^d) Wire(ri,pe,d)
12 (** CtrlSS **)
13 agent CtrlSS = (\s0,s1,a) (^p0,p1) (
   Life(s1,s0,p1,p0) | Attach(a,p0,p1) )

```

136 A Process Descriptions from the Case-Study in MWB/ABC

```

14 agent Life = (\sx,sy,px,py) sx(m) . (~r) (
    Dist(px,m,r) | r . Life(sy,sx,py,px) )
15 agent Dist = (\p,m,r) ( 'p<m> . Dist(p,m,r) + 'r . 0 )
16 agent Attach = (\a,p0,p1) a(c0,c1,cd) . (~d) ( cd(m)
    . 'd<m> . 'd<m> . 0 | Wire(p0,c0,d) |
    Wire(p1,c1,d) | t.Attach(a,p0,p1) )

```

Agents *CtrlIfsP* and *CtrlIfsR* represent distinct parts of process abstraction *CtrlIfs* to describe a component’s behaviour related to its control interfaces for referencing its provided and binding its required functional interfaces, respectively. Agents *CtrlEIP* and *CtrlEIR* represent distinct parts of process abstraction *CtrlEI* to describe a component’s behaviour related to its control interfaces for binding external provided to internal required functional interfaces and for binding external required to internal provided functional interfaces, respectively. Both *CtrlIfs* and *CtrlEI* have been defined in Section 5.2.2.

A.2 Core Behaviour of the Components

The following agents describe core behaviour¹ of the primitive components, as it has been defined in Section 9.5.

```

1  (** OutCore **)
2  agent OutCore = (\poResult,roDone,roReply)
    Out2Core(poResult,roDone,roReply)
3  agent Out2Core = (\poResult,roDone,roReply)
    poResult(res,id) . 'roDone<id> . (
    'roReply<res,id> . 0 |
    t.Out2Core(poResult,roDone,roReply) )
4  (** CtrCore **)
5  agent CtrCore =
    (\pcDone,pteExecTest,rteAttach,rdetachTest,rstopTest,
    rprovRefEInt,rprovRef0Res)
    Ctr2Core(pcDone,pteExecTest,rteAttach,rdetachTest,
    rstopTest,rprovRefEInt,rprovRef0Res)
6  agent Ctr2Core =
    (\pcDone,pteExecTest,rteAttach,rdetachTest,rstopTest,
    rprovRefEInt,rprovRef0Res) pteExecTest(ts,ret) .
    ts(r2stopTest,r2startTest,c,r2,p2) . 'rstopTest .
    'rdetachTest .
    'rteAttach<r2stopTest,r2startTest,rdetachTest> .
    r2(p2bindTInt,p2bindTRes) .
    p2(p2provRefExecuteWithID) . (~ret2)
    'rprovRefEInt<ret2> . ret2(eInteract) .
    'p2bindTInt<eInteract> . 'rprovRef0Res<ret2> .

```

¹ The core behaviour does not include behaviour of default control actions (see Section 5.2.5).

A.3 Complete Behaviour of the Subcomponents 137

```

ret2(oResult) . 'p2bindTRes<oResult> .
'p2provRefExecuteWithID<ret2> .
ret2(p2executeWithID) . 'r2startTest . ((~id)
'ret<id> . 'p2executeWithID<id> . 'id . 0 |
pcDone(id2) . id2 .
Ctr2Core(pcDone,pteExecTest,rteAttach,rdetachTest,
r2stopTest,rprovRefEInt,rprovRef0Res) )
7 (** EnvCore **)
8 agent EnvCore = (\peInteract) Env2Core(peInteract)
9 agent Env2Core = (\peInteract) peInteract(ret) . (
  (~val) 'ret<val> . 0 | t.Env2Core(peInteract) )
10 (** TestCore **)
11 agent TestCore =
  (\pexecuteWithID,rtInteract,rtResult)
  pexecuteWithID(id) . (~ret) 'rtInteract<ret> .
  ret(val) . 'rtResult<val,id> . 0
12 (** TestCoreEquiv **)
13 agent TestCoreEquiv =
  (\pexecuteWithID,rtInteract,rtResult)
  pexecuteWithID(id) . (~comm) ( (~ret)
  'rtInteract<ret> . ret(val) . 'comm<val> . 0 |
  comm(res) . 'rtResult<res,id> . 0 )
14 (** TestCoreNonequiv **)
15 agent TestCoreNonequiv =
  (\pexecuteWithID,rtInteract,rtResult)
  pexecuteWithID(id) . (~ret) 'rtInteract<ret> .
  ret(val) . (~resid) 'rtResult<val,resid> . 0

```

A.3 Complete Behaviour of the Subcomponents

Complete behaviour of the primitive subcomponents can be described by the following agents.

```

1 agent Out = (\s0,s1,pgoResult,psodone,psoreply)
  (~poResult,roDone,roReply) (
  CtrlIfsP(poResult,pgoResult) |
  CtrlIfsR(roDone,psodone) |
  CtrlIfsR(roReply,psoreply) |
  OutCore(poResult,roDone,roReply) )
2 agent Env = (\s0,s1,pgeInteract) (~peInteract) (
  CtrlIfsP(peInteract,pgeInteract) |
  EnvCore(peInteract) )
3 agent Test =
  (\s0,s1,pgexecuteWithID,pstInteract,pstResult)
  (~pexecuteWithID,rtInteract,rtResult) (
  CtrlIfsP(pexecuteWithID,pgexecuteWithID) |
  CtrlIfsR(rtInteract,pstInteract) |

```

138 A Process Descriptions from the Case-Study in MWB/ABC

```

    CtrlIifsR(rtResult , pstResult) |
    TestCore(pexecuteWithID , rtInteract , rtResult) )
4 agent Ctr =
    (\s0 , s1 , pgcDone , pgteExecTest , rteAttach , rdetachTest , rstopTest ,
    rprovRefEInt , rprovRef0Res) (^pcDone , pteExecTest) (
    CtrlIifsP(pcDone , pgcDone) |
    CtrlIifsP(pteExecTest , pgteExecTest) |
    CtrCore(pcDone , pteExecTest , rteAttach , rdetachTest , rstopTest ,
    rprovRefEInt , rprovRef0Res) )

```

For simplicity, we do not include agents describing the components’ behaviour that is related to their control interfaces for cloning the components (see Section 5.2.4 and constant applications of $Ctrl_{clone}$ in Section 9.5). These interfaces are not used.

A.4 Behaviour of the Composite Component

The following agents describe complete behaviour of the composite component, as it has been defined in Section 9.5.

```

1 (** TEcomp **)
2 agent TEcomp = (\s0 , s1 , pgexecuteTest , psasyncReplET)
    (^pexecuteTest , rteExecTest , psteExecTest , rasyncReplET ,
    pteReply , pgteReply , pteAttach) (
    CtrlIifsP(pexecuteTest , pgexecuteTest) |
    CtrlIifsR(rasyncReplET , psasyncReplET) |
    CtrlEIP(pexecuteTest , rteExecTest) |
    CtrlEIP(pteReply , rasyncReplET) |
    CtrlIifsR(rteExecTest , psteExecTest) |
    CtrlIifsP(pteReply , pgteReply) |
    CtrlSS(s0 , s1 , pteAttach) |
    TE2comp(pteAttach , psteExecTest , pgteReply) )
3 agent TE2comp = (\pteAttach , psteExecTest , pgteReply)
    (^sctr0 , sctr1 , sout0 , sout1 , senv0 , senv1 ,
    pgcDone , pgeInteract , pgoResult , pgteExecTest , psoDone , psoReply ,
    rdetachTest , rprovRefEInt , rprovRef0Res , rstopTest , rteAttach)
    (
    Ctr(sctr0 , sctr1 , pgcDone , pgteExecTest , rteAttach , rdetachTest ,
    rstopTest , rprovRefEInt , rprovRef0Res) |
    Out(sout0 , sout1 , pgoResult , psoDone , psoReply) |
    Env(senv0 , senv1 , pgeInteract) | (^d)
    'pteAttach<sctr0 , sctr1 , d> . 0 | (^d)
    'pteAttach<sout0 , sout1 , d> . 0 | (^d)
    'pteAttach<senv0 , senv1 , d> . 0 | (^d)
    Wire(rprovRefEInt , pgeInteract , d) | (^d)
    Wire(rprovRef0Res , pgoResult , d) | (^d)
    Wire(rteAttach , pteAttach , d) |
    TestPlug(rdetachTest , rstopTest) | (^ret)'

```



```

pgteExecTest<ret> . ret(pteExecTest) .
'psteExecTest<pteExecTest> . 0 | (^ret)
'pgteReply<ret> . ret(pteReply) .
'psoReply<pteReply> . 0 | (^ret) 'pgcDone<ret> .
ret(pcDone) . 'psoDone<pcDone> . 0 )
4 agent TestPlug = (\rdetachTest,rstopTest) (
rdetachTest . 0 | rstopTest . 0 )

```

Analogously as in Section A.3, we do not include an agent describing the component’s behaviour that is related to its control interface for cloning the component. This interface is not used.

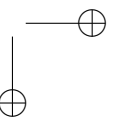
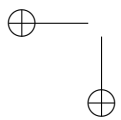
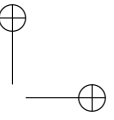
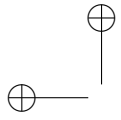
A.5 Services of SOA

Finally, the following agents describe behaviour of the services according to Section 9.4. Process abstraction TE_{init} is included „in-line“ within agent TE.

```

1 (** TM **)
2 agent TM = (\st,fe,lr) (^s) ( TMst(st,fe,s) |
TMar(lr,s) )
3 agent TMst = (\st,fe,s) st(test,ret) . (^r1,r2)
'fe<r1> . r1(et2,ar2) . 'et2<test,r2> . ( r2(id) .
'ret<id> . 0 | 's<ar2> . 0 | t.TMst(st,fe,s) )
4 agent TMar = (\lr,s) ( s(ar2) . ar2(res,id) .
'lr<res,id> . 0 | t.TMar(lr,s) )
5 (** TEB **)
6 agent TEB = (\pe,fe) (^q) ( TEBpub(q,pe) |
TEBfind(q,fe,pe) )
7 agent TEBpub = (\t1,pe) pe(i,d) . (^t2) ( 't1<t2,i,d>
. 0 | t.TEBpub(t2,pe) )
8 agent TEBfind = (\h,fe,pe) h(h2,i,d) . (
t.TEBfind(h2,fe,pe) | ( 'fe<i> . 'pe<i,d> . 0 + d
. 0 ) )
9 (** TE **)
10 agent TE = (\et,ar,pe) 'pe<et,ar> . TEimpl(et,ar)
11 agent TEimpl = (\et,ar) (^s0,s1,ars,etg) ( 'ars<ar>.0
| (^d,t1) 'etg<t1> . t1(p) . Wire(et,p,d) |
TEcomp(s0,s1,etg,ars) )
12 (** TL **)
13 agent TL = (\lr,rl) (^s) ( TLlr(lr,s) | TLrl(rl,s) )
14 agent TLlr = (\lr,t1) lr(res,id) . (^t2) (
't1<t2,res,id> . 0 | t.TLlr(lr,t2) )
15 agent TLrl = (\rl,h) h(h2,res,id) . rl(ret) .
'ret<res,id> . TLrl(rl,h2)
16 (** System **)
17 agent System = (\st,rl) (^et,ar,lr,pe,fe) (
TM(st,fe,lr) | TE(et,ar,pe) | TL(lr,rl) |
TEB(pe,fe) )

```



Název	Modelling of Component-Based Systems with Mobile Architecture
Autoři	Mgr. Marek Rychlý, Ph.D. doc. Ing. Jaroslav Zendulka, CSc.
Vydavatel	Vysoké učení technické v Brně Fakulta informačních technologií
Obálka	Mgr. Dagmar Hejduková
Tisk	MJ servis, spol. s r.o.
Vyšlo	Brno, 2010
Vydání	první

Tato publikace neprošla redakční ani jazykovou úpravou.