University of Duisburg-Essen
Institute for Computer Science and Business Information Systems
Working group “Specification of Software Systems”
Prof. Dr. Michael Goedicke

Applying Formal Component Specifications to Module Systems in Java

Master’s thesis

Applied Computer Science - Systems Engineering
Software Systems Engineering

Marco Müller
<marco.mueller@uni-due.de>
Spichernstraße 24
45138 Essen
Matriculation number: 2213373

Essen, 26th March 2010

Supervisor: Dipl.-Wirt.-Inf. Moritz Balz
Reviewer: Prof. Dr. Michael Goedicke

Abstract

Formal component models and component-based software engineering have been subject to research for decades. The resulting component models allow for systematic development of large software systems, while usually focusing on a small set of aspects of an architecture. The formal founding of some languages permits reasoning about architecture attributes. However, current programming languages and component frameworks hardly reflect such features with respect to hierarchical architectures, context independence, and behavioural descriptions of components. Thus the frameworks do not leverage the benefits of the profound research of component-based software engineering, like enhanced understandability, formal reasoning about quality attributes and ease of maintenance. Instead of creating loosely coupled, self-describing, and self-contained software components, the frameworks tend to use tightly coupled modules. This thesis first deals with an analysis of existing formal component models with different foci. As a result, a set of desirable feature categories for component models are identified. These categories are related to OSGi, a module framework that is considered to be integrated into the Java platform. The results of this comparison are the foundation for a proposal for a new component model in OSGi. The reference implementation of the proposal is evaluated in a case study, which shows the applicability of the proposed model. Finally the thesis discusses the challenges that came up in the case study and addresses issues needed to be solved for a productive implementation of the component model.

Contents

Figures

Listings

Abbreviations

ADL	Architecture Description Language
API	Application Programming Interface
CBSE	Component-based Software Engineering
CEM	Concurrently Executable Module
CORBA	Common Object Request Broker Architecture
EJB	Enterprise Java Beans
HTTP	Hypertext Transport Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
ISO	International Organization for Standardization
JAR	Java Archive
JAXB	Java Architecture for XML Binding
JEE	Java Enterprise Edition
JSR	Java Specification Request
KLAPER	Kernel Language for Performance and Reliability analysis
MOF	Meta Object Facility
MVC	Model-View-Controller
OMG	Object Management Group
PCM	Palladio Component Model
POJI	Plain Old Java Interface
RDSEFF	Resource Demanding Service Effect Specification
RMI	Remote Method Invocation
ROBOCOP	Robust Open Component Based Software Architecture for Configurable Devices Project
RPC	Remote Procedure Call
SEFF	Service Effect Specification
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

This chapter describes the motivation and the problem statement of the thesis, along with an introduction in component-based software engineering (CBSE) [Szy02] and why it is still not adopted in practical software engineering. In chapter 1.2 basic concepts of CBSE are shown and related to the concepts used in frameworks usually referred to in a component context. The thesis is motivated in chapter 1.1 and an outline of the document is given in chapter 1.3.

1.1 Motivation

The idea of decomposing problems into smaller parts in a process commonly called "divide and conquer" is a mature idea which has also been subject of research in Software Engineering since the early 1970s. Parnas' work on criteria for the decomposition of systems [Par72] is one of the first scientific publications related to this topic in computer science.

Complexity is an important problem, especially in software engineering, as the systems built today tend to be large projects, having many connections with other complex systems. Breaking down these projects into manageable parts is a key to handle this complexity not only for initial development, but also for maintenance. In addition these manageable parts can be reused in other software projects. Reused parts are usually well-understood and improvements on these reused parts may be of benefit to each product using them.

One approach for reducing complexity with "divide and conquer" in software engineering is CBSE. In CBSE a system is decomposed into loosely-coupled components. Components are independently deployable units that offer a functionality to be called by other components and may call other components' functionality themselves. The offered functionality is usually called a *provision*, while necessary calls to other components' functionality are usually referred to as *requirement*. In CBSE the software architecture consists of components that are interconnected through their provisions and requirements. These interconnected components represent the program. When components are to be reusable in other contexts, they must provide generic functionality.

Components communicate with each other using well-defined interfaces which describe the functionality of the underlying component. Connectors between components are used to interconnect the requirements of one component with the provisions of another component. Usually, components can be basic components, providing their functionality in terms of executable code, or composite components, which provide their functionality by instantiating and interconnecting subcomponents. These composite components may also be subcomponents of a composite, thus creating a hierarchical architecture. Such hierarchical architectures provide different levels of abstraction by hiding the subcomponents of a composite. These multi-level architectures enhance the understandability of the system by hiding unnecessary information. Components are usually black-box entities, hiding their implementation details as proposed in Parnas' information hiding principles [Par72]. A thorough description of component models is given in chapter 2 in this thesis.

Software systems developed using a component-based approach have many advantages over monolithic software architectures:

1. Software components can be distributed for parallel development.

2. Maintenance of software components is easier, because changes in the component implementation can be made locally, without effects on the complete system, as long as the interfaces are not changed.

3. If components are loosely coupled, single components can be exchanged with new ones.

These advantages especially take effect in the development and maintenance of large systems. [Sam97, Chap. 1]

The component models developed as a result of the thorough research do not only facilitate a structured development of modular systems, but are often also formally founded [AG97, CS01, CFGGR91, MDEK95, SG94]. This formal foundation allows for verification of system characteristics, specification of component interaction mechanisms, and simulation of the system before it is implemented.

However, while research on component models is very advanced, current programming languages, platforms, and frameworks hardly reflect the related concepts. Considering modern programming languages like Java [GJSB05] or C# [ISO06], component definitions are optional and limited to namespaces. Namespaces provide means to combine sets of classes that semantically belong together. The highest level of abstraction in modern programming languages are classes and their relations. Thus external frameworks are needed to provide component definitions and functionality.

The features provided by these frameworks do not leverage the functionality proposed by formal component models long ago. Thus the practice-driven component frameworks and platforms cannot benefit from the advanced research of formal component models.

This thesis addresses the gap between the state of research and practice-driven frameworks by comparing the concepts of formal component models with the features of a framework for the Java language and platform. The widely-used OSGi framework [OSG09a] is used for comparison.

The contribution of this thesis is a proposal for a change of the OSGi Service Platform to provide the features of formal component models. This proposal is implemented in this thesis and its functionality is evaluated in a case study.

1.2 Basics

Formal component models have a variety of features and foci, including component interconnection [AG97], message flow [CS01], data abstraction and concurrency [CFGGR91], dynamic architectures [MDEK95, BHP06], or modeling and prediction of quality attributes [GMRS08, RBH+07, Inf03].

These component models share features like compositionality as well as required and provided interfaces between components. Figure 1.1 shows a component in formal component models schematically. The appended circle represents a provided interface as a contract defining how to use the component. The semicircular appendix is a required interface, describing what the component requires from its context. The rectangle is used to describe a component body. The component in this example includes interconnected subcomponents implementing the features provided by the component.

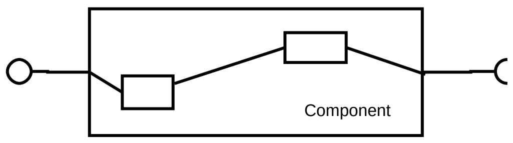


Figure 1.1: A schematic representation of a component in formal component models with required and provided interfaces as contracts, and subcomponents.

The component's functionality can be accessed via the interfaces. In formal component models, these interfaces, the component itself, and the interconnection of components is defined formally, permitting e.g. a verification of the component interconnection.

Practice-driven component frameworks like OSGi, the Enterprise Java Beans (EJB) [Sun09c] in the Java Enterprise Edition (JEE) [Sun09b], or the Common Object Request Broker Architecture (CORBA) Component Model [Obj06] lack some of the features of formal component models. Figure 1.2 shows components usually found in practice-driven component frameworks. Hierarchical component architectures are not supported, as the concept of composite components is not implemented. Additionally, components state their requirements to their context by directly referencing the provided interfaces of other components, hence employing class level dependencies that have to be resolved at compile time. These components are not context independent, as the desired context has to be available for the component's compilation.

1.3 Thesis Outline

Chapter 2 will introduce formal component specifications by presenting a selection of specification languages and their included component models. The concepts are then compared to identify similarities and differences between the approaches. Chapter 3 gives an overview of the OSGi Service Platform. The framework is explained and its component model is compared to the essential features of formal component

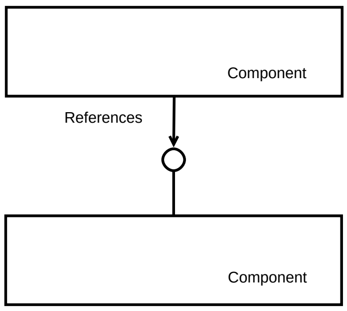


Figure 1.2: In practice-driven frameworks, component structures are usually flat, i.e. no subcomponents are available. They also do not explicitly state their requirements to their context, but directly reference the provisions of other components.

specifications. In chapter 4 a component model for OSGi is described, which was developed and implemented in this thesis. The proposed component model aims at closing the gap between the formal component specifications and OSGi. Chapter 5 describes the evaluation of the proposal using a case study. A discussion in chapter 5.2 discovers the strengths and the weaknesses of the proposed component model. Chapter 6 discusses related work before the thesis is concluded in chapter 7.

2 Formal Component Specifications

Modular architectures are one of the key concepts for managing the complexity of large software systems. Thus many approaches for describing modular architectures exist. In this chapter, the fundamentals of component-based software engineering (CBSE) are introduced first, before different architecture description languages (ADL) are examined, which are used for describing software components and their communication. For this purpose, ADLs with different foci are considered: SOFA 2 [BHP06] focuses on dynamic architectures, Palladio [RBH^{+}07] and KLAPER [GMS05] on the modeling and prediction of quality requirements. UniCon's [SDK^{+}95] aim is to be universally applicable and Pi [SG94] focuses on data abstraction and concurrency. UML is also considered, though it is not a formal component specification, but a widely-used means for describing software architectures.

2.1 Component-based Software Engineering

The term software components is not defined unambiguously. Hopkins integrates several definitions in [Hop00]. His definition is: *"A software component is a physical packaging of executable software with a well- defined and published interface."*. Szyperski's "Compoent Software" [Szy02], which is one of the foundational works on CBSE, contains three different definitions: (1) *"Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system."* [Szy02, Preface]; (2) *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."* [Szy02, Chap. 4.1.5]; and (3) *"A component is a set of normally simultaneously deployed atomic components"*, with the following definition of atomic components: *"An atomic component is a module and a set of resources"*. In the last definition, modules and resources are: (1) *"A module is a set of classes and possibly non-object-oriented constructs, such as procedures or functions"* and (2) *"A resource is a 'frozen' collection of typed items."* [Szy02, Chap. 20.3]. Many more definitions exist, as can be seen in [BDH^{+}98].

Despite the lack of an unambiguous definition, some component characteristics can be derived, for they are repeatedly described: A component is a black-box entity, which is reusable in different contexts without any need of knowledge about the component's internals or modification of those. To enable reusability, the components have to be self-describing, i.e. they need to provide publicly visible, well-defined interfaces. Components consisting of interconnected subcomponents form component hierarchies, which ultimately form the application [BHP06].

2.2 Palladio Component Model

The Palladio Component Model (PCM) [RBH+07] is an architecture language with focus on performance prediction. It has been mainly developed at the University of Oldenburg and the Karlsruhe Institute of Technology. The PCM is implemented with the Eclipse Modeling Framework (EMF) [Ecl10a], and is thus based on the EMF/Ecore meta model. Systems in Palladio are developed using a rich graphical modeling software software called *PCM-Bench* [Kar09].

In Palladio a system consists of components communicating with shared interfaces. Components can be composite to form a hierarchical architecture. The PCM-Bench can simulate a system's architecture. Processing nodes and links between them can be defined as simulated hardware, and components can be allocated to these resources. Additionally, a usage model can be defined which describes a synthetic load for a simulation with estimated user behaviour. Due to this simulation, performance issues in the architecture can be identified before the system is implemented.

2.2.1 Interface Definition

Components in Palladio define required interfaces to describe which functionality is required by other components, and provided interfaces to describe functionality that is offered to the context. Messages are shared using operation calls on these shared interfaces. Communication via message flow, unix-pipes etc. is not considered. In the PCM interfaces can be defined at three levels of details:

- Signature List Based Interface
  The interface consists of method signatures, which are comparable with operation signatures of programming languages like Java. The operations have a return value, a name, in, out and inout parameters as well as exception types.

- Protocol Enhanced Interface
  The protocol enhanced interfaces define permitted sequences of operation calls. PCM does not define the syntax of the protocol, thus different concepts like finite state machines or petri-nets may be used. For performance evaluation, the protocols used in the interfaces need to be compatible.

- Quality of Service Enhanced Interface
  For annotating interfaces with properties and constraints regarding the quality of service, they can be attributed using the Resource Demanding Service Effect Specification (RDSEFF) [RBH+07].

2.2.2 Component and System Definitions

Components in Palladio are active or passive units of computation with provided and optionally required interfaces. The PCM defines four component types, in three levels of abstraction, which represent three phases in the development time of an architecture. The component type hierarchy is shown in figure 2.1.

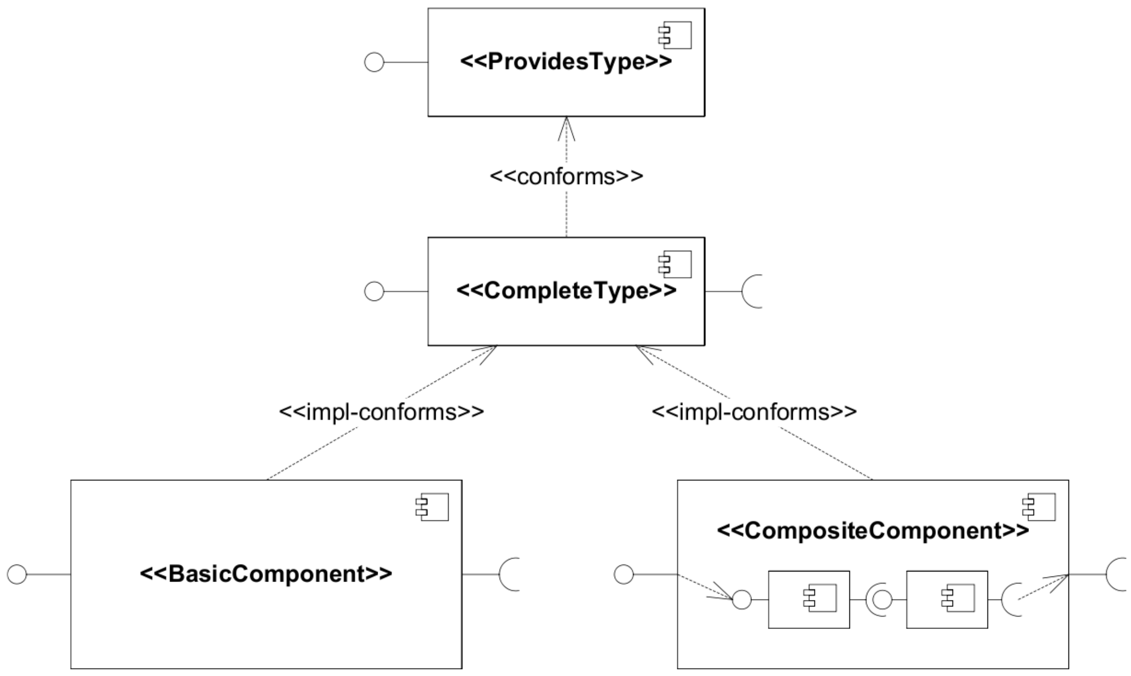


Figure 2.1: Component type hierarchy in Palladio [RBH+07]

- Provides Type
  The provides type is the most abstract type and just contains the information necessary for receiving the provided functionality. The requirements of the component are not considered at that development phase. As the possible requirements of the component are not defined yet, the description needs to be refined to the *CompleteType* in a later phase.

- Complete Type
  Components of the complete type specify their provided functionality as the provides type does. Additionally, their requirements are defined. This type is used in a later phase of the development, when the complete architecture with all transitive requirements is considered.

- Implementation Type Components
  Implementation type components are a refinement of complete type components. They additionally contain implementation details, describing how their functionality is implemented. Implementation type components may be *composite components* which consist of interconnected subcomponents, and *basic components*, describing their functionality by a Service Effect Specification (SEFF). The SEFF is a behavioural description similar to UML activity diagrams [Obj09b], but it just considers behaviour concerning the required or provided roles or resource consumption of the hardware underlying the components, i.e. CPU cycles, hard disk times, etc.

A system in Palladio is defined in a separate diagram. In this diagram components are instantiated and the instances are interconnected. The requirements and provisions of component instances may be delegated to the systems context, thus rendering the system a composite component itself.

2.2.3 Deployment and Simulation

As the focus of Palladio is performance prediction, the PCM requires means to describe a deployment environment. For this reason, a resource environment can be defined. The resource environment has resource containers, which are nodes containing e.g. processors and hard disks, and linking resources, which represent interconnections between the resource containers, and can be used to represent e.g. network connections. The resource demand of basic components is given with an abstract value, due to the possibly changing underlying hardware. The CPU load can e.g. be stated in cycles. The resource demand may be given not only as a constant, but also as complex functions, to cover probabilistic loads and dependencies on parameters like the size of input data.

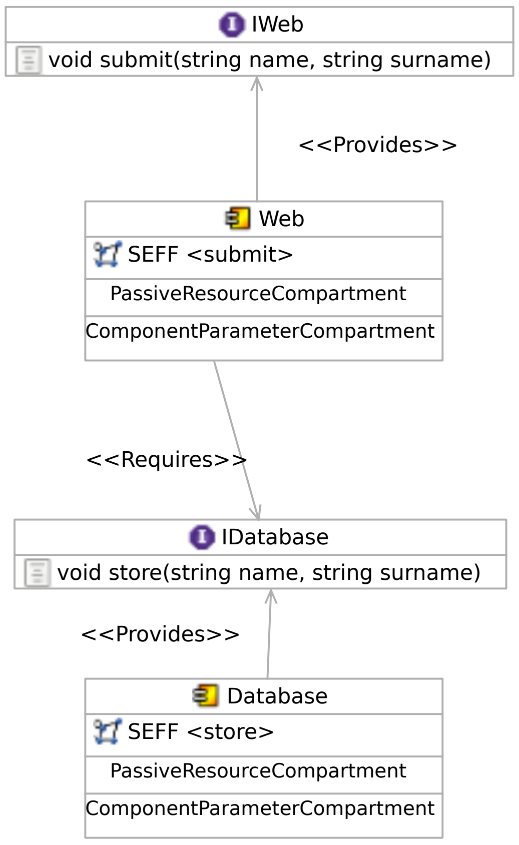
A separate allocation diagram is used to allocate the components of a system to resource containers in a resource diagram. It is also possible to define a usage model, modeling actions of users interacting with the system and thus describing a workload. With this information the architecture can be simulated to evaluate performance issues in the system. As this thesis focuses on the static architecture description, the simulation will be omitted here.

2.2.4 Example

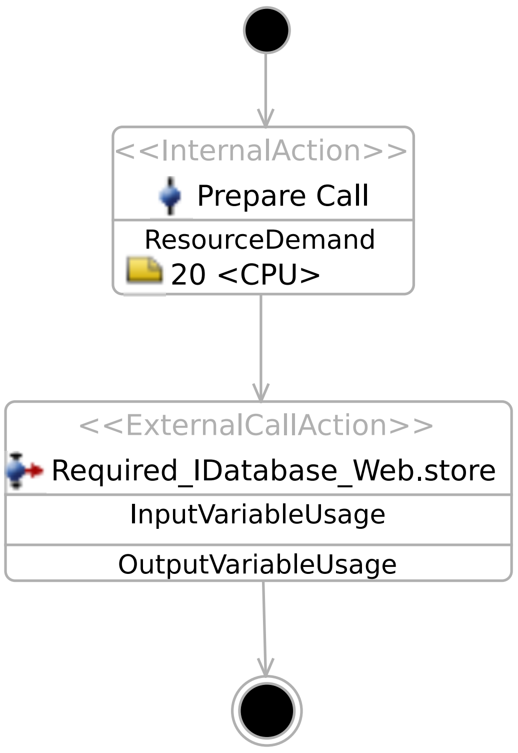
Figure 2.2a shows an example of a repository in the PCM-Bench. A repository stores the defined interfaces and components of an architecture. The exemplary repository includes two interfaces, *IWeb* and *IDatabase*. *IDatabase* and *IWeb* each declare one operation with two parameters. The PCM-Bench does not permit to identify in, out, or inout parameters. Although the component model defines three levels of detail for interface description, the PCM-Bench does not support Protocol and Quality of Service Enhanced Interfaces.

The exemplary repository depicted in figure 2.2a defines two components, *Web* and *Database*, which both are basic components. The component *Web* provides the interface *IWeb* and requires the interface *IDatabase*. Its implementation is given using a SEFF, which is defined in figure 2.2b. The component *Database* provides the interface *IDatabase*. The SEFF in figure 2.2b defines the implementation of the operation *submit* of the component *Web*. After the operation call, the component performs an internal action, which has a resource demand of 20 CPU cycles. After the internal action, a call of the *store* operation of the required interface *IDatabase* is made before the control flow ends.

Figure 2.3 shows an example of a resource environment that includes two resource containers and one link between them (the connection is not depicted in the figure).

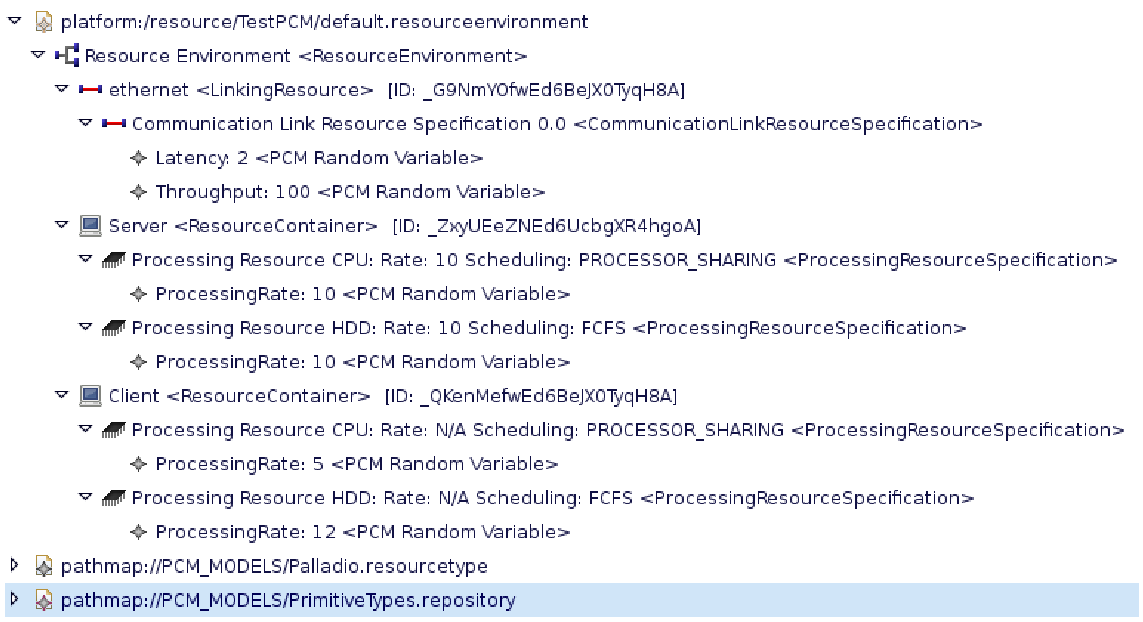

IWeb
void submit(string name, string surname)
<<Provides>>
Web
SEFF <submit>
PassiveResourceCompartment
ComponentParameterCompartment
<<Requires>>
IDatabase
void store(string name, string surname)
<<Provides>>
Database
SEFF <store>
PassiveResourceCompartment
ComponentParameterCompartment

(a) Example of a Palladio repository



(b) Example SEFF in Palladio

Figure 2.2: Figure (a) shows an exemplary repository in Palladio. The component *Database* provides the interface *IDatabase*. The same interface is required by the component *Web*, which also provides the interface *IWeb*. In figure (b) the SEFF of the component *Web* is shown. The first action of the SEFF is an internal action with a resource demand of 20 CPU cycles. The next step models a method invocation *store* of the required interface *IDatabase*, before the execution of the SEFF is finished.



```
platform:/resource/TestPCM/default.resourceenvironment
  Resource Environment <ResourceEnvironment>
    ethernet <LinkingResource>  [ID: _G9NmYOfwEd6BeJX0TyqH8A]
      Communication Link Resource Specification 0.0 <CommunicationLinkResourceSpecification>
        Latency: 2 <PCM Random Variable>
        Throughput: 100 <PCM Random Variable>
    Server <ResourceContainer>  [ID: _ZxyUEeZNEd6UcbgXR4hgoA]
      Processing Resource CPU: Rate: 10 Scheduling: PROCESSOR_SHARING <ProcessingResourceSpecification>
        ProcessingRate: 10 <PCM Random Variable>
      Processing Resource HDD: Rate: 10 Scheduling: FCFS <ProcessingResourceSpecification>
        ProcessingRate: 10 <PCM Random Variable>
    Client <ResourceContainer>  [ID: _QKenMefwEd6BeJX0TyqH8A]
      Processing Resource CPU: Rate: N/A Scheduling: PROCESSOR_SHARING <ProcessingResourceSpecification>
        ProcessingRate: 5 <PCM Random Variable>
      Processing Resource HDD: Rate: N/A Scheduling: FCFS <ProcessingResourceSpecification>
        ProcessingRate: 12 <PCM Random Variable>
pathmap://PCM_MODELS/Palladio.resourcetype
pathmap://PCM_MODELS/PrimitiveTypes.repository
```

Figure 2.3: The exemplary resource environment contains two resource containers (Server and Client) and a link between them.

2.3 The Π language

The Π language [SG94] is a formal textual component specification and interconnection language with a focus on distributed systems, especially data abstraction and concurrency, and the incremental development of software systems [CFGGR91, SG94]. In Π a software component is considered an autonomous unit of computation which can be concurrently executed. Thus the components are called Concurrently Executable Modules (CEM).

A CEM consists of four sections as shown in figure 2.4: *import*, *export*, *common parameters* and the *body*. The export and import sections formally describe the types to be imported from or to be exported to other CEMs, including their operations. The body provides the implementation of the export expressed in the imported data types and operations. The body may also introduce new data types and operations to realize the export. The common parameters section describes imported properties which are also exported, thus publishing some information about the import through the export interface.

Π provides a view concept consisting of a *type view*, an *imperative view*, a *concurrency view* and a *type connection view*. The type view describes the static properties of the data types of a CEM, i.e the properties independent from execution. The imperative view expresses the operations specified in the type view in an imperative manner, thus showing how a request is executed including possible side effects of the operation. In the concurrency view, the import and the export sections are described with respect to concurrency.

Unlike most other ADLs, the import of a CEM is not an interface defined separately and shared by all CEMs in the system, but is directly contained in the

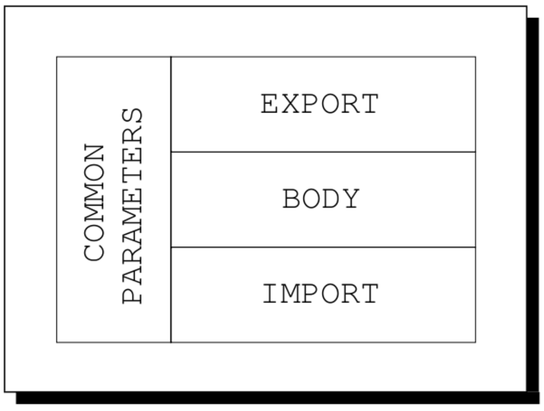


Figure 2.4: A CEM in Π consists of four sections. Import, export and common parameters define the interfaces, while the body defines the implementation, which is hidden to the CEM's context. [SG94]

component description. The import is a formal description of the required interface in terms of types, including their behaviour and concurrency constraints. For this reason, the type connection view is needed to define a mapping of import, export and common parameters sections of CEMs. This view is used to construct composite CEMs and, finally, complete system architectures.

2.3.1 Component Interface Specification

The export, import and common parameters sections are described by the type view, the imperative view and the concurrency view. Each view is mandatory.

The Type View

The data types required, used, and provided by the CEM are described in the type view. A type is defined by a name, a set of operation signatures and a set of invariant properties expressed in equations. Optionally, informal descriptions can be provided.

Listing 2.1 shows the definition of the type *Car* in the export section, thus defining a provided data type that can be used by other CEMs. The type *Car* has five operations. As an example, the operation *setLocation* is a function to be called on a *Car* object. The method invocation returns a *Car* object. The method takes a parameter of the type *Location*, called *loc*. The type *Location* has to be defined in either the body, the import, or the common parameters section to be used in this definition. Another operation is called *getLocation*. It is to be called on a *Car* object and also returns a *Car* object. The equation describes the invariant that the operation *getLocation* called on a *Car* object with a preceding *setLocation* call will return the location given as parameter in the *setLocation* call. Types in the import, common parameters, and body section are defined in the same way.

```
type view specification
    export
```

```
        type Car
            general description { A car may be used for driving }

            operation start : Location -> Car
            operation stop : Car -> Car
            operation isStarted : Car -> Boolean
                equations
                    isStarted(start(loc)) = true;
                    isStarted(stop(start(loc))) = false;

            operation setLocation : Car -> Car
                variables loc : Location

            operation getLocation : Car -> Location
                equations
                    getLocation(setLocation(loc)) = loc
```

Listing 2.1: Example of the type view on an exported type in Π

The Imperative View

While the type view describes operations on types as functions, it does not define the possible side effects of an operation. These can be expressed in the imperative view on a CEM.

In the imperative view, the operations of a type are expressed in a syntax similarly known from imperative programming languages like Java or Interface Definition Languages (IDL) like the OMG IDL [Obj08]. Each operation has a name, a return value, and parameters. A parameter is described by its direction (in, out, or inout), a name and a type.

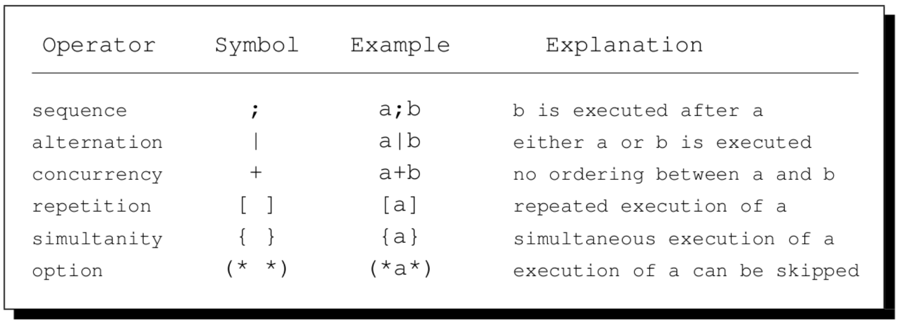
As the type *Car*, defined in listing 2.1, exports an operation that uses the type *Location* as a variable, CEMs that use this type also need access to a CEM implementing the type Location. Thus listing 2.2 shows the imperative view on a type in the common parameters section. The type *Location* is imported from the context and exported to the context again. The types *String* and *Tuple_2* used in this listing are also to be described by the body, the import or the common parameters section of this CEM, as the Π language itself does not provide any types.

```
imperative view specification
    common parameters
        type Location
            operation getName() : String
            operation getCoordinates() : Tuple_2
```

Listing 2.2: Example of the imperative view on a common parameter type in Π

The Concurrency View

The type view and the imperative view specify the functionality of types in a CEM. This presumes that the operations are allowed to be executed. The concurrency view is used to define a sequence of operation calls that is permitted on an instance of a type, including concurrent calls. Path expressions [See87] over operation names are used to formally state which concurrency constraints imported operations must fulfill or, respectively, which concurrency constraints the exported operations have. These path expressions are essentially regular expressions using operation names as words, extended by operations regarding concurrency and simultaneity. Figure 2.5 shows the list of operators of path expressions. Additionally, preconditions about states can be given for operations in the concurrency view. These preconditions relate to an observable state of the CEM.



Operator	Symbol	Example	Explanation
sequence	;	a;b	b is executed after a
alternation	|	a|b	either a or b is executed
concurrency	+	a+b	no ordering between a and b
repetition	[ ]	[a]	repeated execution of a
simultanity	{ }	{a}	simultaneous execution of a
option	(* *)	(*a*)	execution of a can be skipped

Figure 2.5: List of operators of path expressions in Π [SG94]

Listing 2.3 shows the concurrency view on the exported data type *Car*. The method *start* must be invoked first. After this invocation, optionally and repeatedly *setLocation* may be executed, or simultaneously (i.e. concurrently, unlimited times in parallel) *getLocation* or *isStarted* is permitted. At last, the method *stop* must be called. Alternatively to this sequence, *isStarted* may be called simultaneously. Path expressions are implicitly repeatable as a whole, thus a car can always be started after stopping.

Preconditions are used in this example to describe the permission for execution depending on internal states. In listing 2.3 preconditions are defined for the *start* and *stop* operations. They are only permitted to be executed when the operation *isStarted* returns a specified value.

```
concurrency view specification
    export
        type Car
            path expression
                ( start ;
                    (* [ setLocation | { getLocation } | { isStarted } ] *);
                    stop )
                | { isStarted }

            precondition definition list
                precondition of start is not(isStarted)
                precondition of stop is isStarted
```

Listing 2.3: Example of the concurrency view on an exported type in Π

2.3.2 Component Implementation

The implementation of the exported types in Π is defined in the body section of a CEM. For this purpose, the body can use the imported types and define its own operations. Components may be atomic or composite. Atomic components implement their functionality directly, while the body of composite components consists of a configuration of subcomponents. Basic components are the third type of components, besides atomic and composite components. These components are described by their interfaces, as their implementation is considered a black box.

Atomic Components

The body of atomic components is described by the type view or – alternatively – the imperative view. In Π a component implements exactly one data type. Thus the body section is used for the construction of this one type, using imported types or internal definitions. The component's type view is notated using the same algebraic specifications that are used for describing the exported type. This enables the developer to verify the compatibility of the export and its implementation. Listing 2.4 shows an excerpt of the algebraic implementation of the exported type *Car*. The operation *create_car_tuple* is an operation of an imported data type which stores two values, a boolean for the car being started or not, and a *Location* object.

```
type view specification
    body
        construction of type Car is CarImplAlgebraic

        operation start : Car -> Car
            variables loc : Location
            equations
                start(loc) = create_car_tuple(true, loc)

        operation getLocation : Car -> Location
            equations
                getLocation(create_car_tuple(true, loc)) = loc

        [...]
```

Listing 2.4: An excerpt of an algebraic implementation of a component in Π.

Alternatively, the implementation of types can be specified in the imperative view, using a language with the features of higher object-oriented programming languages. Listing 2.5 shows an excerpt of the imperative implementation of the exported type *Car*.

```
imperative view specification
    body
        construction of type Car is CarImplImperative

        operation start(in loc : Location) returns Car
            begin
               return create_car_tuple(true, loc);
            end

        operation getLocation(in car : Car) returns Location
            description
               { The value is stored in the second
                  position of the tuple, with a
                  starting index of 1. }
            begin
               return getValue(1, create_car_tuple(true, loc)) = loc
```

Listing 2.5: An excerpt of an imperative implementation of a component in Π.

Composite Components and System Assembly

Composite components consist of interconnected instances of subcomponents. Instantiation of components is called *incarnation* in Π. The implementation of composite components is also called *configuration*. A configuration contains a set of incarnations of subcomponents, their interconnections, and the interconnection between the specification of the subcomponents and the specification of the parent composite component. As a composite component is a component itself, it also has exported and imported types as well as common parameters. These are connected to the according sections of subcomponents. Thus the composite component delegates the operation calls to its subcomponents. A system in Π is a composite component that is not embedded into another composite component.

An example for a configuration in Π is shown in listing 2.6. In this example, two components are incarnated first. Each component incarnation has a name. In a second step, the components within the configuration are interconnected. In this case, *Location* – the imported type of *Car* – is connected to the export of a type called *PointOfInterest*, which has not been introduced in the examples above. In the last step, the export of the composite component is connected to the export of the *Car* type.

```
configuration CarManagement
   component incarnations
      car : Car;
      poi : PointOfInterest;

   component interconnections
      connections of car
         from poi import
            type Location <- PointOfInterest

            operations
               getName <- getPoiName;
               getCoordinates <- getGeoLocation;

      connections of Export
```

```
        from Car import
           type Car <- Car

           operations
              start <- start;
              stop <- stop;
              isStarted <- isStarted;
              getLocation <- getLocation;
              setLocation <- setLocation;
end configuration CarManagement
```

Listing 2.6: A configuration in Π

2.4 SOFA 2

SOFA 2 [BHP06] is a component system mainly developed by the Distributed Systems Research Group at the Charles University of Prague. Its focus includes dynamic reconfiguration of architectures and different communication techniques. The component model of SOFA 2, which is described in [ČHPR09], is based on a well-defined meta-model. The meta-model is described by the Meta Object Facility (MOF) technology, which is also the basis for the Unified Modeling Language (UML). A system configuration in SOFA 2 is expressed in the Extensible Markup Language (XML). The components are implemented in Java.

The architectural elements in SOFA 2 are components and connectors which are interconnected using shared interfaces. The components are represented to their context by so called *frames*. The frame is a black-box view of a component, which defines its provided and required interfaces. A so-called *architecture* of a component provides the component's implementation details. The architecture either directly implements a frame (primitive component), or composes subcomponents to implement the frame.

2.4.1 Frame and Interface Definition

The component frame defines the borders of the component and specifies the required and provided interfaces. The component content is the implementation of the component and may be executable code or an architecture of subcomponents. A frame may be implemented by many components, thus the components implementing the same frame are exchangeable with each other. At run time, the control part represents the component and the control interfaces allow access to component meta data. An abstraction of a SOFA component is shown in figure 2.6.

Listing 2.7 shows an example of the definition of a simple SOFA 2 interface. The interface has the name *auth* and a reference to a Java interface. The Java interface defines the functionality offered by the SOFA interface.

```
<itf-type name="IAuthentication"
       signature="org.example.IAuthentication" />
```

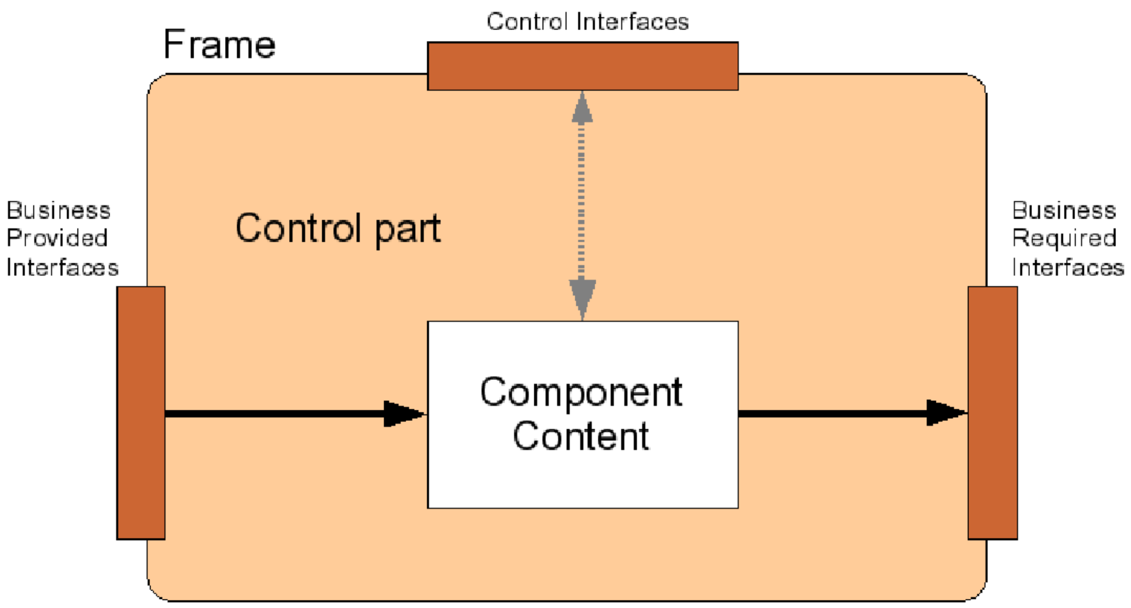


Figure 2.6: Components in SOFA are represented by a *control part* at runtime. The frame forms the component border and describes provided and required business interfaces which define the provided and required behaviour. Control interfaces are also available to dynamically control the life cycle of the component. The component content may be a direct implementation of the component behaviour or be composed of subcomponents. [ČHPR09]

Listing 2.7: Interface definition in SOFA 2

The interface may be used by different component frames. The sample frame in listing 2.8 has the interface defined in listing 2.7 as a required interface. The *comm-style* attribute of the *requires* and *provides* tags define the communication style supported by this frame for the given interface. As frames are design time units, rather than run time units, they are not implemented in the underlying programming language.

```
<frame name="AuthenticationFrame">

    <requires name="users"
        itf-type="sofatype://IUsersDB"
        comm-style= "method_invocation" />

    <provides name="auth"
        itf-type="sofatype://IAuthentication"
        comm-style= "method_invocation" />

</frame>
```

Listing 2.8: Frame definition in SOFA 2

2.4.2 Connector and Architecture Definition

Connectors in SOFA 2 [BP04] are, besides components, first class entities. The SOFA component model leverages four communications styles for component interaction: synchronous operation calls, asynchronous message delivery, uni- and bidirectional data streams, and communication using shared memory. However, the shared memory communication style is not supported by the SOFA 2 runtime, though the language allows for custom communication styles to be added by implementing them in Java.

Architectures in SOFA 2 may be primitive, which means they implement the behaviour of a frame directly, or composite, using subcomponents. A primitive architecture is defined by specifying a name, the frame of the architecture, and the implementation. A composite architecture needs to define subcomponents, connectors, and connections of an implementation instead. A composite architecture with required or provided interfaces could itself be subject to composition.

Figure 2.7 shows an abstraction of a simple SOFA 2 architecture with connectors. Components 1 and 3 are composite components, while component 1 consists of the components 2 and 3, and component 3 in turn consists of the components 4 and 5. Component 1 is the whole application. The darker boxes on the side of the components represent the required and provided interfaces of the components. The arrows are connectors between the interfaces. The source of the arrow is the required interface and the interface the arrow is pointing at is the provided interface (component 4 to component 5) . In case of a connection between a parent component and its subcomponent, the arrow represents a delegation or requirements or provisions (e.g. component 3 to component 4). The delegation indicates that the requirement or provision of a subcomponent is also the requirement or provision of the parent composite.

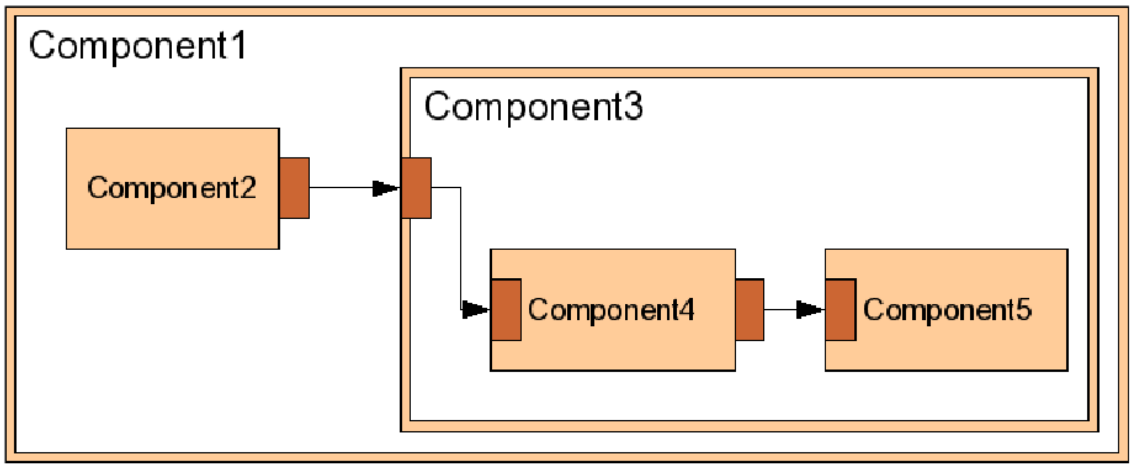


Figure 2.7: Components in SOFA 2 can be composite (Component 1 and 3) or primitive (2, 4 and 5). Component 1 is the whole application. The darker boxes represent required and provided interfaces of the component. The arrows are connectors providing communication between the components. [ČHPR09]

Examples

In listing 2.9, a primitive architecture is defined. The architecture is implemented by the Java class *org.example.LDAP_Authentication*.

```
<architecture name="LDAP_Authentication"
    frame="sofatype://AuthenticationFrame"
    impl="org.example.LDAP_Authentication" />
```

Listing 2.9: A primitive architecture in SOFA 2

In listing 2.10, a composite architecture is defined, which uses the architecture *LDAP_Authentication* defined in listing 2.9 as a subcomponent. The frame of *AuthenticationTest* is a simple, empty frame without any interfaces. It is the top-level component of the application.

```
<architecture name="AuthenticationTest"
    frame="sofatype://AuthenticationTestFrame">
    <sub-comp name="ldap_auth"
        frame="sofatype://AuthenticationFrame"
        arch="sofatype://LDAP_Authentication" />
    <sub-comp name="testDataSource"
        frame="sofatype://LoginTestDataSourceFrame"
        arch="sofatype://LoginTestDataSource" />
    <connection>
        <endpoint sub-comp="testDataSource" itf="auth"/>
        <endpoint sub-comp="ldap_auth" itf="auth"/>
    </connection>

</architecture>
```

Listing 2.10: A composite architecture in SOFA 2

In this example, the architecture *AuthenticationTest* instantiates the components *LDAP_Authentication* and *LoginTestDataSource*. *LDAP_Authentication* provides the interface *IAuthentication*, which is stated as required interface by the component *LoginTestDataSource* (not shown in the examples). The composite architecture can thus interconnect these components using the shared interface.

The composite architecture in this example is a top-level component and thus neither requires nor provides interfaces.

2.5 UML Composition Diagram

The Unified Modeling Language (UML) [Obj09a, Obj09b] is a language developed by the Object Management Group (OMG) and standardised by the International Organization for Standardization (ISO) as ISO/IEC 19501. The current version of the UML is 2.2. The UML is not an ADL in terms of the other languages presented here, but a set of notations which may also be used to represent software architectures. While the language allows for different structural and behavioural descriptions of systems, the capabilities for architecture description are focused on here.

The specification of UML mainly consists of semi-formal diagrams and informal text, as the semantics of UML is not formally defined. Despite this fact, the UML is widely-used for architecture description and will thus be considered here. In general, the UML provides different facilities for the notation of elements. The specification defines the abstract syntax of the language, but not the concrete syntax. For simplicity reasons, the graphical notation that is used in the UML specification document will also be used here.

2.5.1 Interface and Component Definition

UML interfaces are defined as an own entity, consisting of a name and a list of operation signatures. Components in UML are entities containing arbitrary content as implementation details. The implementation may e.g. be described using UML class diagrams and UML state diagrams, rendering them to be simple components. Composite components are insofar different from simple components as their implementation consists of internal component instances and their interconnection. A system in UML is also modeled as a composite component.

Interfaces are attached to components using interaction points called *ports*. The ports reference well-defined interfaces to be provided or required. They describe the structural aspects of the interaction by stating required and provided functionality.

The component owning the port can access the environment using the port's required interfaces. The environment can also access the component only through the interfaces provided by its ports. The set of provided interfaces attached to a port is called the *type* of a port. Additionally, behavioural constraints can be given for single interfaces, ports, or components. The definition of behavioural constraints is mentioned by the specification, but not explained.

2.5.2 Connection Definition

Components are interconnected through their provided and required interfaces. A required interface of a component is assembled with a provided interface of another component through assembly connectors. Cyclic dependencies are not allowed. When connected, the requiring component can use the provided functionality. For the connector to be applicable, the involved required and provided interfaces must be compatible, i.e. the provided interface must offer the same or more operations as the required interface.

For interconnecting a component with its parent composite, the *delegates* connector is used. In a composite the provided interfaces are implemented by the internal components. The delegates connector is used to provide the functionality that an internal component of the composite provides. The delegates connector is also used to export one or more required interfaces of internal components as an required interface of the composite component.

2.5.3 Graphical Notation and Example

An interface is notated as a rectangle with two parts. the upper part contains the key word *Interface* and the interface's name. The lower part contains the list of operation signatures. The operation signature is similar to those known of current programming languages like Java or C#. An exemplary interface definition is shown in figure 2.8 at the right side.

Components are also notated as a rectangle. This rectangle contains the key word *component* and the component's name, and alternatively as a graphical representation of the key word *component* the icon shown in the upper right corner of the component in figure 2.8 at the left side. Both representations of the key word may coexist. A component's port is notated as a small rectangle attached to the component. A port has a name, which should be written next to it. A reference to a required or provided interface is notated with the ball and socket notation, as to be seen in figure 2.8. The provided interface is *Voting*, and the required interface is *Database*. The definition of the interface *Database* is not included in this example.

Figure 2.9 shows an example of an composite component in UML. The composite component *Store* has a provided interface *OrderEntry* and a required interface *Account*. It has interconnected, internal component instances. Component instances are notated as components with a colon in front of their name. The assembly connector, which interconnects a requirement with a provision, is notated with as the ball and the socket engaged, as shown in figure 2.9 between the components *Order* and *Customer*. The component instance *Order* implements the provided interface

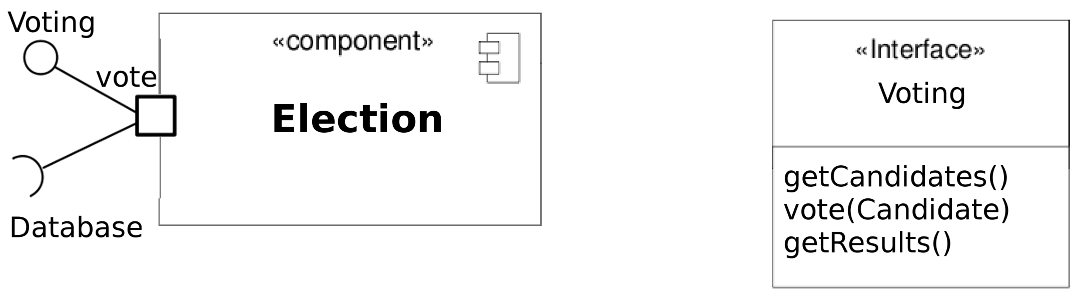


Figure 2.8: On the left side a component called *Election* is represented with a port and two interfaces, a provided interface *Voting* and a required interface *Database*. On the right side, the interface *Voting* is defined.

OrderEntry, indicated by the delegate connector, which is notated with an arrow. The required interface *Account* originates from the internal component instance *Customer* and is exported as a required interface of the composite component.

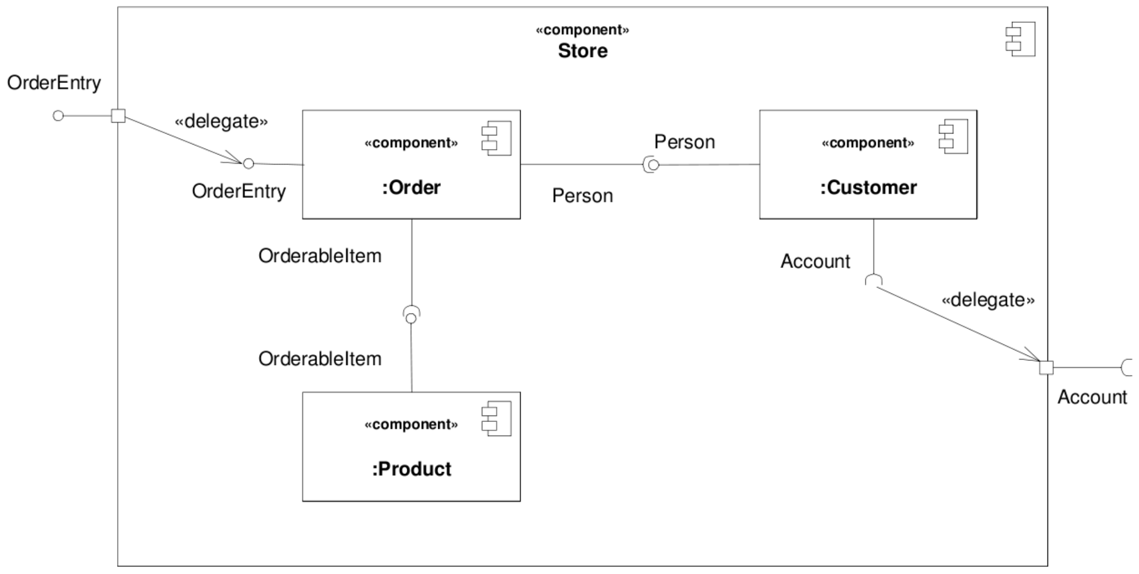


Figure 2.9: The store is a composite component, which embodies three component instances. The store has a required and a provided interface which are delegated to or from internal components. [Obj09b]

2.6 UniCon

UniCon is an ADL invented by Shaw et al. [SDK+95] with the focus on offering functionality and entities that were widely used by software architects at the time the language was developed, but were not supported by a language and according tools.

The main elements of a UniCon system are components and connectors. Both define the implementation and specifications shown in figure 2.10. The specification has a type and a unit of association. The associative units of components are players. Connectors use ports as associative units. These associative units are used for interconnecting components using connectors.

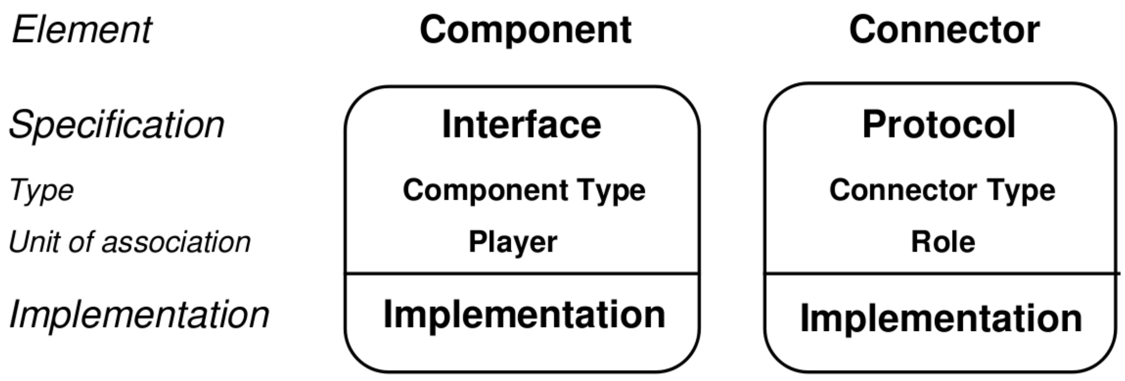


Figure 2.10: The main architecture elements in UniCon are components and connectors. Both consist of a specification, which has a type and a unit of association, and an implementation. [SDK+95]

2.6.1 Component Definition

Components are units of computation. They are specified by an interface definition, its type and a set of players. An interface is an instance of a type. A type constrains the set of possible players for that interface. Players are used for connections between components and connectors, and are explained in section 2.6.3.

Components may be primitive or composite. Primitive and composite components can be distinguished by their implementation. The implementation of primitive components is specified with an implementation type. Implementation types are e.g. *source*, indicating that a source code file represents the implementation, or *executable*, stating an executable file to be the implementation of the primitive component. Implementation types may also require attributes, e.g. parameters to pass to an executable. New implementation types may also be introduced by the developer.

Composite components define a configuration within their implementation section by instantiating and interconnecting components using connectors. Subcomponents can be primitive or composite components as well. This structure allows for hierarchical component systems.

As UniCon doesn't validate the referenced implementation file representing a component, the programmer has to ensure that the implementation complies with the specification.

2.6.2 Connector Definition

Connectors are definitions for the relations of components among each other. They enable components to communicate. Connectors are not necessarily explicitly represented by source code, but may be a shared memory or more complex communication mechanism like Remote Procedure Call (RPC).

Connectors are specified by a protocol, its type, a set of roles. Similarly to the component interface type, the connector protocol type constrains the set of possible roles of a protocol. Roles are used for connections between connectors and components, and are explained in section 2.6.3.

The implementation of a connector is always *BUILTIN*, defining the connection to be Unix pipes. *BUILTIN* is the only implementation type supported by the language. Also, while composite connectors are generally supported by the component model, they are not supported by the language.

2.6.3 Component Interconnection

Components and connectors are represented to their context by their specification. I.e., other elements can use them without knowing the implementation. With this property, the elements in UniCon follow Parnas' information hiding principles [Par72]. The type of a specification defines the supported type of *players* or *roles* for that component. UniCon provides a predefined set of types for components and connectors. Components and connectors are interconnected using the *players* and *roles* as shown in figure 2.11. The type of a *player* and an associated *role* must match for the configuration to be valid.

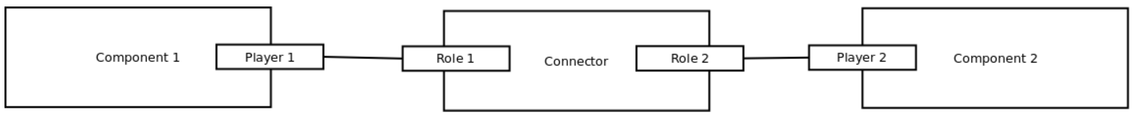


Figure 2.11: A component in UniCon has one or more *players*, which are its points of interconnection. The *players* are connected to matching *roles* of connectors to enable communication.

UniCon defines 14 player types, which represent provided and required services. Interfaces can provide services in terms of operations (*RoutineDef*), data (*GlobalDataDef*), output operations as system calls (*WriteFile*), sequential file writing (*WriteNext*), writing in Unix Pipes (*StreamOut*) and remote procedure calls (*RPCDef*). They can also require services accordingly with the player types *RoutineCall*, *GlobalDataCall*, *ReadFile*, *ReadNext*, *StreamIn* and *RPCCall*, which are described in detail by Shaw et al. in [SDK$^+$95]. The player types *PLBundle* and *RTLoad* differ from the other types, as *RTLoad* enables to give information required for real-time scheduling of the component and *PLBundle* can bundle a collection of operation and data players.

A component type is used to express the intention of the component's provided functionality. The number, types, and specification of players for a component is restricted by the component type. As an example, the component type *Filter* supports the two player types *StreamIn* and *StreamOut*.

Connectors may have one of seven built-in connector types. Each connector type restricts the number, type and specification of roles for a connector, similarly to the component types.

Each role type accepts one or more player types. As an example, the connector type *Pipe* defines the role types *Source* and *Sink* to be supported. The role type *Source* accepts the player type *StreamOut* of the component type *Filter*, and *ReadNext* of *SeqFile*. The role type *Sink* accepts the player type *StreamIn* of the component type *Filter*, and *WriteNext* of *SeqFile*. Thus a connector of the type *Pipe* can be used to interconnect two components of the type *Filter*. Role types may be attributed with additional information as parameters. Detailed tables of attributes, role and connector types, as well as player and component types can be found in [SDK+95].

2.6.4 System Configuration

A system in UniCon is a composite component without players. A composite component is defined by its parts, its configuration, and an abstraction. The parts are instantiations of components and connectors, which constitute the composite component. The configuration is the association of component players with connector roles, that interconnect components and connectors. The abstraction specifies how the players of the interface are associated to the players of the implementation. As systems in UniCon are made up of composite components, the configuration of a system and the configuration of subsystems are the same.

2.6.5 Example

Listing 2.11 shows the definition of a primitive component called *stack*. The component contains an interface definition and the implementation part. The implementation refers to the source file *stack.c*, a source code file written in C. The interface of the component has the type *Computation*. The type *Computation* supports the player types *RoutineDef*, *RoutineCall*, *GlobalDataUse* and *PLBundle*. In this example, the interface has one player of the type *PLBundle*. The members of the *PLBundle* are four *RoutineDefs*, each given with a name, the player type and the signature of the provided operation. The signature is a list of parameters and its syntax depends the programming language.

```
COMPONENT stack
    INTERFACE IS
        TYPE Computation
        PLAYER stackness IS PLBundle
            MEMBER (init_stack; RoutineDef; SIGNATURE (; "void"))
            MEMBER (stack_is_empty; RoutineDef; SIGNATURE (; "int"))
            MEMBER (push; RoutineDef; SIGNATURE ("char␣*"; "void"))
            MEMBER (pop; RoutineDef; SIGNATURE ("char␣*"; "void"))
        END stackness
    END INTERFACE

    IMPLEMENTATION IS
        VARIANT stack IN "stack.c"
            IMPLTYPE (Source)
        END stack
    END IMPLEMENTATION
END stack
```

Listing 2.11: A sample component in UniCon [SDK+95]

The example in listing 2.12 shows the declaration of a primitive connector called *Unix-pipe*. The connector in this example has the type *Pipe*.

Additionally, but not presented in this example, protocols may include assertions that constrain the entire connector, like rules about timing or ordering. These assertions are defined as property lists within the protocol definition.

The primitive connector in example in listing 2.12 has the type *Pipe* and specifies the roles *Source* and *Sink*. Both roles are constrained with the optional attribute *MAXCONNS*. This attribute constrains the maximum number of players, that this role can be bound to. The implementation of the connector is set to *BUILDIN*, which specifies the connector to be a primitive connector and its implementation to be Unix pipes.

```
CONNECTOR Unix-pipe
   PROTOCOL IS
      TYPE Pipe
      ROLE source IS source
         MAXCONNS (1)
         END source
      ROLE sink IS sink
         MAXCONNS (1)
         END sink
   END PROTOCOL
   IMPLEMENTATION IS
      BUILTIN
   END IMPLEMENTATION
END Unix-pipe
```

Listing 2.12: A sample connector in UniCon [SDK+95]

Listing 2.13 is an example of a system with a display showing the output of a random number generator. The components communicate using the Unix-pipe declared in listing 2.12. The composite component *mySystem* instantiates the subcomponents and interconnects the players with the roles to form a system. The system has no players itself. It could however provide and require behaviour by defining players in its interface. Some of the internal roles would be connected to the composite component's players then, which defines the communication of the context with the internal components. This would render the composite component a subsystem that can be used in a greater context.

```
COMPONENT randomNumberGenerator
    INTERFACE IS
        TYPE Filter
            PLAYER output IS StreamOut
                SIGNATURE ("line")
                PORTBINDING (stdout)
            END output
    END INTERFACE

    IMPLEMENTATION IS
        VARIANT randomNumberGenerator IN "rng.c"
            IMPLTYPE (Source)
        END stack
    END IMPLEMENTATION
END randomNumberGenerator

COMPONENT display
    INTERFACE IS
        TYPE Filter
            PLAYER input IS StreamIn
                SIGNATURE ("line")
                PORTBINDING (stdin)
            END input
    END INTERFACE

    IMPLEMENTATION IS
        VARIANT display IN "display.c"
            IMPLTYPE (Source)
        END stack
    END IMPLEMENTATION
END display

COMPONENT mySystem
    INTERFACE is
        TYPE General
    END INTERFACE

    IMPLEMENTATION IS
        /* Instantiate the random number generator,
            the display and the connector Unix-pipe. */
        USES rng INTERFACE randomNumberGenerator
        USES display1 INTERFACE display

        USES P PROTOCOL Unix-pipe

        /* Interconnect the internal elements */
        CONNECT rng.output TO P.source
        CONNECT display1.input TO P.sink
    END IMPLEMENTATION
```

Listing 2.13: A sample system in UniCon

2.7 KLAPER

KLAPER (Kernel LAnguage for PErformance and Reliability analysis) is an intermediate language for performance and reliability analysis of component-based systems [GMS05]. It has been developed mainly at the Università di Roma and the Politecnico di Milano. The focus of KLAPER is not to design a component-based system, but to be an intermediate language for transforming a component-based design into a language used for performance and reliability analysis. The main idea of KLAPER is that component-based systems are designed using languages that are well suited for software design, but that for system analysis other languages are to be preferred. With KLAPER, the transformation from the different design languages to the different analysis languages is simplified, as it is not necessary to find transformation rules from each design language to each analysis language, but just to find transformation rules from each design language to KLAPER and from KLAPER to each analysis language, as shown in figure 2.12. KLAPER is not an ADL like the other languages presented here, but for representing the architecture designed in an ADL, KLAPER has to provide concepts that are common to all ADLs. With this focus, KLAPER should provide all features of component models considered necessary by its authors.

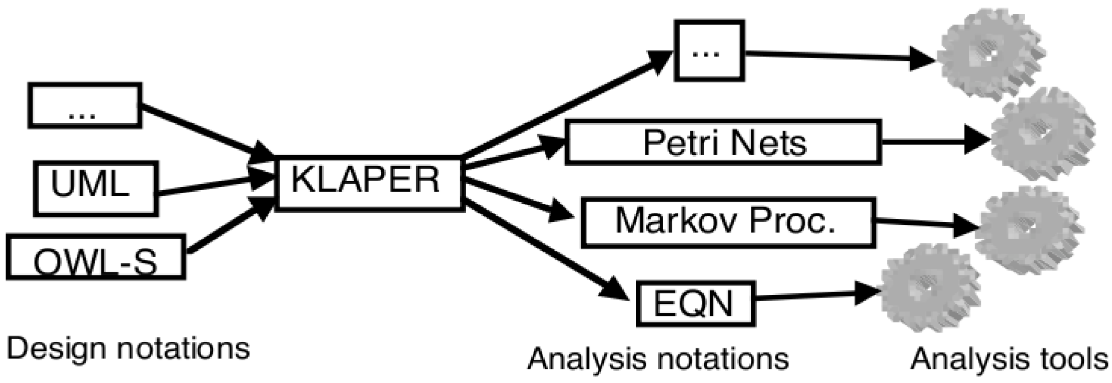


Figure 2.12: KLAPER is an intermediate language for transforming design languages to analysis languages. The component-based design languages on the left can be transformed (semi-)automatically into a KLAPER representation. Afterwards, the KLAPER model can be transformed (semi)automatically into a representation in the analysis language, which can then be used to analyse the designed system. [GMS05]

KLAPER is a MOF-based language, just as SOFA 2 presented in chapter 2.4. Thus KLAPER can be expressed with diagrams similar to UML class diagrams or e.g. in XML. In KLAPER a system consists of interacting *resources*, with resources providing and possibly requiring *services*. Resources are not limited to software components but may also represent physical resources like processors or communication links, which enables the language to represent analysable models.

2.7.1 Component Definition and Assembly

A component in KLAPER is defined by a resource offering a service. The implementation of the service is described with a behaviour. Resources represent logical (software components) or physical units (processors, communication links, ...). They have attributes, including a name, a type and performance properties. The services offered by a resource have a name and formal parameters with which they can be invoked. Thus a service represents one operation. The behavior resulting from the implementation of the service is represented by a control flow of steps, including a start, an end, control elements like forks and joins, as well as internal activities and service calls, as to be seen in figure 2.13. A *Behaviour* is thus a sequence of *Steps* with a defined start and a defined end. *InternalActivities* are steps of computation of the resource implementing the service, while *ServiceCalls* are calls to required services. The complete meta model of KLAPER is depicted in figure 2.13.

When all necessary components are defined and the system is to be assembled, service calls are associated with the corresponding service definitions. As different instances of one component type may exist, the direct association allows for a unique distinction between single component instances.

Figure 2.14 shows an example component in KLAPER. The resource *aSortComp* represents a component of the type *sortComp*. The component offers a service called *sort*, which can be called with the two parameters *lstin* and *listout*, which are both integer parameters.

The service is implemented by a short control flow, which consists of just one service call to a service called *process* of a component with the type *cpu*. The service takes one parameter, which depends on a value called *list*. This is actually the value of *lstin*, a formal parameter of the service *sort*.

As can be seen in the example above, KLAPER also has the capability to define performance and reliability attributes for resources, services, and behaviour. This fact is not considered here, as this thesis focuses on the functional aspects of the component model.

2.8 Comparison Summary

After inspecting the given component models, the features provided by the languages can be categorised into eight areas, which are described in more detail in the following sections. Table 2.1 gives an overview of the comparison results.

2.8.1 Provided Interfaces

As large systems in CBSE are divided into smaller components, each component offers services that may contribute to resulting systems. Since single components

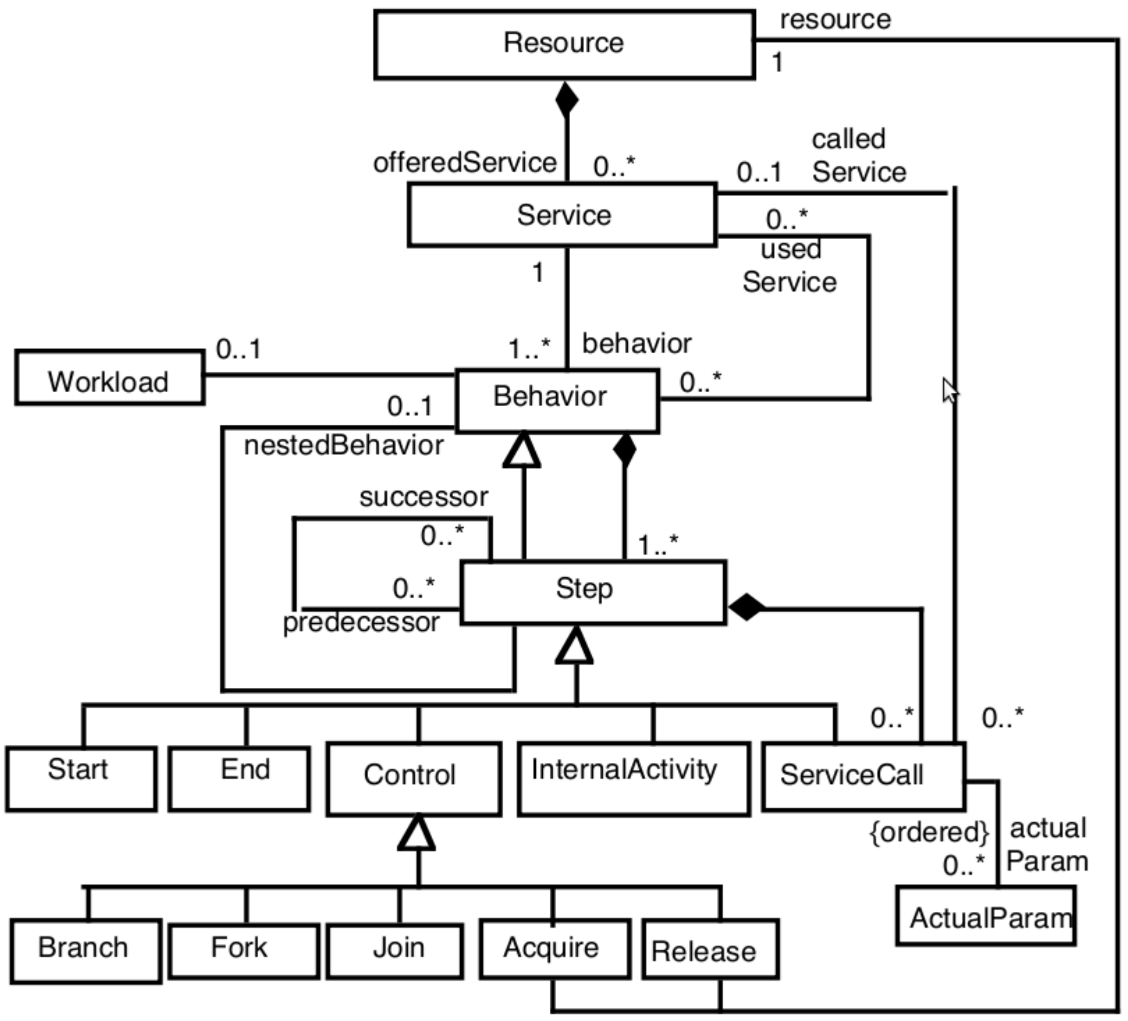


Figure 2.13: In the KLAPER meta model, a *Resource* is a logical (software component) or physical (processor, communication link, . . .) unit which offers *Services*. Services are implemented by one or more *Behaviors*, which in turn possibly require services. The attributes of the entities are not included in this view. [GMS05]

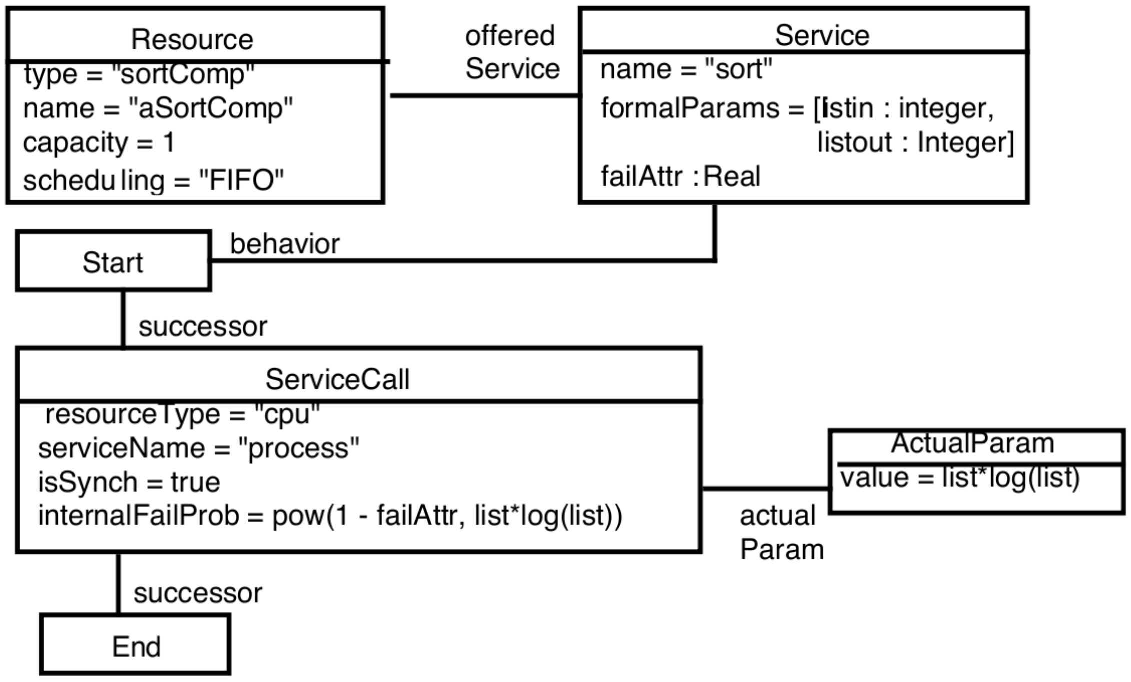


Figure 2.14: The component *aSortComp* in this example is defined by its *Resource*, an offered *Service* and its corresponding *Behaviour*. The service is realised by a three internal steps, including *start*, *stop* and a service call to a resource of the type *cpu*, which has to be defined and assembled later. [GMS05]

Features of the inspected ADLs						
Provided Interfaces	√	√	√	√	√	√
Dependencies	√	√	√	√	√	√
Composition	√	√	√	√	√	x
Connectors	√	√	√	x	√	x
Communication Constraints	√	√	x	√	√	√
Instantiation	x	√	√	√	√	x
Assembly	√	√	√	√	√	√
Quality attributes	√	x	x	x	x	√
ADL	Palladio	Π	SOFA 2	UML	UniCon	KLAPER

Table 2.1: Comparison summary of ADL features

may be used in different contexts, which are possibly unknown during development of the component, the interfaces must be described thoroughly. Each of the considered languages provides means for describing the provided interfaces, though at different levels of detail. In Π the export is described in three views, including concurrency constraints and exported types, while in Palladio the description is limited to references to Java interfaces and a communication type.

2.8.2 Dependencies

The usage of services provided by other components introduces dependencies between the components of a system. For dependency descriptions to be context-independent, they must precisely define the functional requirements that can be satisfied by different components. The components describe functional dependencies in all considered languages, though also varying in the degree of detail. In SOFA 2 e.g., only interface names are used to describe required services. In contrast, Π expressed the import in three views, just as its export.

2.8.3 Composition

When systems become larger, the subsystems in focus also grow in size. The concept of component composition simplifies to keep an overview of the complete system, by introducing abstraction layers. All ADLs except KLAPER consider composite components for this purpose. I.e., a set of interconnected components can be defined as one component with provided and required services. The context does not need to distinguish whether a component is primitive or composite. This allows for developing systems of large building blocks. In all languages the provided services of a composite component are delegated to their subcomponents, and its unbound required services are delegated to the composite.

2.8.4 Connectors

Different interconnection types between components exist. Communication may e.g. be event-based or use method invocation. Connectors may become even more complex, when they should ensure quality attributes like security [Szy02, chapter 21.1.2]. Many of the examined languages do not cover the possible complexity of connectors and the resulting effects, including delays and failure probabilities. In Palladio, Π, UML, and KLAPER the only interaction mechanism is method invocation. In SOFA 2 different interaction mechanisms can be applied, but are limited to a predefined set. The component model in UniCon allows for arbitrary connector implementations, but the language implementation is limited to Unix-Pipes. However, complex communication mechanisms can be represented by a connector component, i.e. a component with the functionality of a connector. The communicating components are then not connected to each other, but both are connected with a direct reference to the connector component, as depicted in figure 2.15.

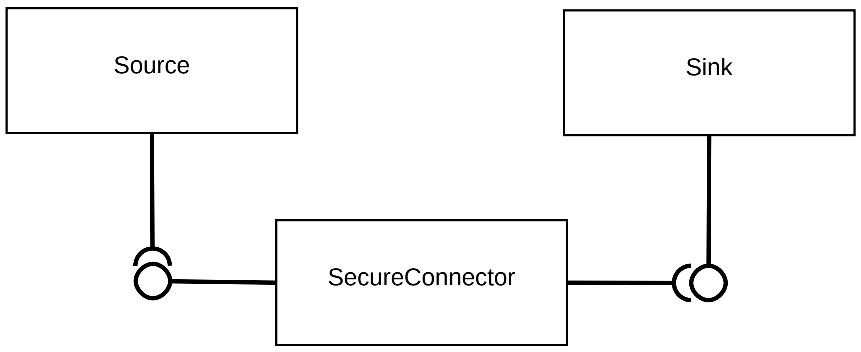


Figure 2.15: A connector component can be used to substitute complex connectors. The component *SecureConnector* is a component that is specifically designed to securely transport the signals from the component *Source* to the component *Sink* over an insecure network.

2.8.5 Communication Constraints

In an open environment, each component may e.g. call each operation of its required service, or each event is allowed. Communication is instead often constrained in terms of call sequences or concurrency. In Π the communication may be constrained using the concurrency view. The constraints include preconditions for operation calls as well as a sequence of possibly concurrent calls using path expressions. UniCon predefines a set of communication constraints for the roles of a protocol. In KLAPER the permitted communication is thoroughly defined using behavioural models. In Palladio and UML communication constraints are mentioned but not defined. SOFA 2 does not consider the interaction at all.

2.8.6 Instantiation

Multiple instances of a component allow for easily reusing components at run time. If an instance of a component fails, components requiring other instances of this component are not affected. For this reason, SOFA 2 defines a system architecture which contains named component instances. Connections are then defined between the component instances. Π, UML, and UniCon use a similar approach. Component instances are not considered in KLAPER and Palladio. In these languages a component must be copied and renamed to simulate an instantiation.

2.8.7 Assembly

In all inspected languages the system is defined in some sort of assembly. At development time, components are defined and implemented independently from their context, while at a separate assembly time the components are interconnected to subsystems and systems. At assembly time, Π, SOFA 2, UML, and UniCon use a top level composite component for instantiating the components and interconnecting them. In Palladio a separate system diagram is created, which represents the top view of the system. While it is described in a separate diagram, it is essentially also a composite component. KLAPER does not provide composite components. Thus an assembly in KLAPER includes all components and their referenced entities in a flat structure, which might be an issue in large architectures.

2.8.8 Quality Attributes

Requirements specifications for large software systems usually also include quality requirements, like performance requirements. For evaluating quality issues at design time, the language must consider these requirements besides the structural design. As performance prediction is one of the goals for Palladio, these quality attributes are easy to specify and allow for evaluating the architecture for performance issues before it is built. In KLAPER the architecture can be attributed with performance and reliability parameters that can be used for analysis after the transformation to applicable analysis models. This functionality cannot be taken for granted. The other examined languages do not provide features for quality requirements.

3 OSGi Service Platform

The OSGi Service Platform (formerly an abbreviation of "Open Services Gateway initiative", now just OSGi) is a practice-driven component framework for the Java language, which was initially introduced as "Java Specification Request (JSR) 291: Dynamic Component Support for Java SE" [Sun07]. Its Core Specification [OSG09a] describes the framework and the component model. The Service Compendium [OSG09b] describes a set of standard components for the framework. The current version of the specification is 4.2. Many implementations of the specification exist. Equinox e.g. is a wide-spread open source implementation of the framework, as well as Apache Felix, both aiming to implement the specification for universal use. Other commercial implementations exist, like the ProSyst mBedded Server, which has extensions (i.e. pre-defined bundles) for smart home, mobile phone, and telematics applications [Pro10].

The goal of the OSGi Service Platform is to provide a middleware for dynamic software architectures, since the Java language and platform do not provide any component concept. For this reason, the framework supports the management of so-called *bundles*, which are technically Java Archives (JAR) containing the compiled classes and corresponding resources. The bundles are configured in the JAR configuration file (`MANIFEST.MF`) using name-value pairs. Bundle meta data include the unique symbolic bundle name and version as well as dependencies to other bundles or functional dependencies. The framework is responsible for loading the bundles, resolving dependencies, and managing the bundle life cycle, as the bundles may be installed and removed at run time.

To achieve its goals, the framework is divided into five layers:

- Module Layer
- Life Cycle Layer
- Service Layer
- Security Layer
- Actual Services

The bundle uses the Module, Life Cycle and Service Layer to provide the actual services, and the bundle communication is constrained by the Security Layer. The architecture is depicted in figure 3.1.

3.1 Module Layer

The Module Layer adds a more abstract view on applications and packages to Java by introducing a modularisation model, the so-called bundles. Bundles are uniquely

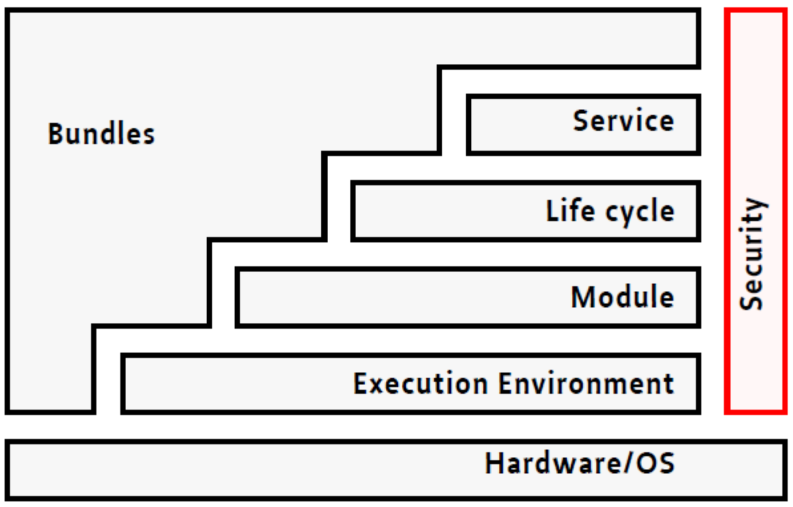


Figure 3.1: Due to the Java Platform, OSGi is independent from hardware and operating systems. The execution environment is the Java runtime platform. Bundles use the Module, Life Cycle and Service Layer to provide the actual services. The Security Layer can be used to constraint the permissions of bundle communication. [OSG09a]

identified by their symbolic name and their version. Each bundle may name packages and classes to be accessible by other bundles, while the rest of the classes is hidden. Bundles requiring functionality from other bundles import packages defined by the package name, which may be exported by an arbitrary bundle. Alternatively, a bundle can declare a dependency to a specific bundle, by referencing the symbolic name of the required bundle. This results in an import of all its exported packages. Resolving dependencies is automatically done by the framework. The automatic resolving process can be controlled descriptively, by naming preferred versions, vendors or bundles to be bound.

The module configuration properties in the JAR configuration file include the following:

- *Activator*
  Bundles have a life cycle, which is shown in figure 3.2. When a bundle is started, the *activator* is executed. The *activator* is a class that inherits from the OSGi specific class *BundleActivator*. The activator has methods which are executed when the bundle is started or stopped, and can store a reference to the bundle's context that the bundle's internals may use.

- *Bundle-Name*
  The human-readable name of the bundle.

- *Bundle-SymbolicName*
  The unique ID of the bundle.

- *Bundle-Version*
  Bundles may exist in different versions, even in parallel at run time. The framework resolves dependencies between components, which may rely on specific versions or version ranges. With this mechanism, bundles can be updated at run time without affecting existing associations, as both versions can run in parallel.

- *Bundle-RequiredExecutionEnvironment*
  The bundle may provide a list of environments it requires to work. These possible environments include the complete Java Standard Edition Runtime Environment in version 6 or special execution environments for limited devices. This is essentially a dependency for a specified context.

- *Export-Package*
  Packages within the bundle that are to be visible to other bundles.

- *Import-Package*
  Packages to be imported by other bundles.

- *Require-Bundle*
  Require-Bundle declares a direct dependency for another bundle. All exported packages of the required bundle are imported.

Example 3.1 shows bundle meta data declaring a bundle with the symbolic name *org.example.persistence* in the version *1.0.0.GA*. It requires the Java Standard Edition in the version 6 as environment and declares a class as activator class. The bundle imports the package *org.osgi.framework*, which is provided by the platform, in version *1.3.0*. The described bundle also exports a package, giving its version number. The version number is an optional parameter.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Persistence Example
Bundle-SymbolicName: org.example.persistence
Bundle-Version: 1.0.0.GA
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Activator: org.example.persistency.Activator
Import-Package: org.osgi.framework;version="1.3.0"
Export-Package: org.example.persistency.export;version="1.0"
```

Listing 3.1: A bundle description in OSGi

3.2 Life Cycle Layer

The Life Cycle Layer defines an application programming interface (API) which allows to install, start, update, stop, and uninstall bundles at run time. As the Life Cycle Layer relies on the existence of bundles, it can not be used without the Module Layer. Figure 3.2 shows the life cycle of bundles.

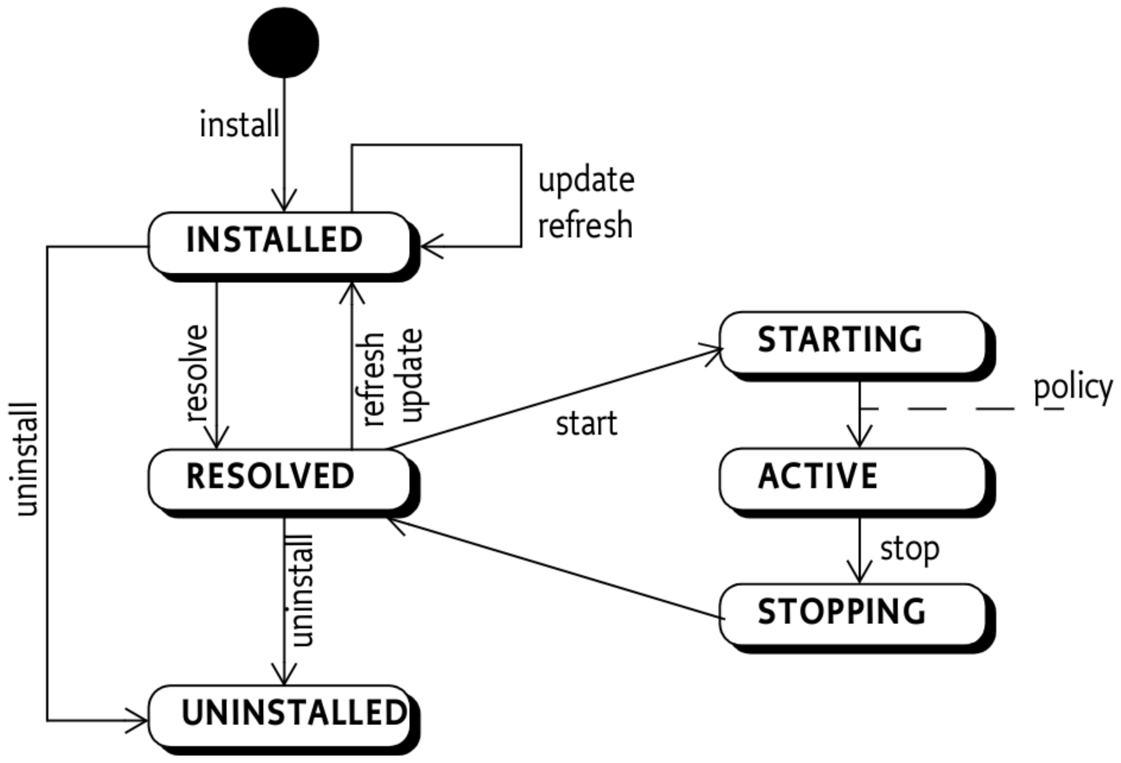


Figure 3.2: The life cycle of a bundle begins with its installation. If all dependencies can be resolved, the bundle is ready to be started. When the bundle is stopped again, it returns to the state *resolved*, where it can be restarted or uninstalled. The uninstallation ends the life cycle of a bundle. [OSG09a]

The life cycle of a bundle begins with its installation in the framework, which causes the bundle's state to switch to *Installed*. In this state, the bundle may be uninstalled, refreshed or updated. When a bundle is refreshed, it is unresolved and the framework tries to resolve the bundle again. An update can be used to replace a bundle with a newer version. The framework automatically resolves all the dependencies of a bundle. Thus if a bundle can be resolved, its state automatically changes to *Resolved*. In this state, the bundle may still be refreshed, updated and uninstalled as in the state *Installed*. Additionally, the bundle can be started, which changes the state to *Starting* and triggers a call to the bundle's activator class. The state is automatically changed to *Active* when the bundle has finished starting. If the bundle is advised to start lazily (via the *Bundle-ActivationPolicy* property), it might also keep the state *Starting* until one of its classes is requested by other bundles. When the bundle is in the state *Active*, it is running and accessible to the other bundles. When the bundle is stopped, the state switches to *Stopping* and triggers a call to the activator class to stop the bundle. The state then eventually switches to *Resolved* again. When a bundle is uninstalled, its life cycle ends.

3.3 Service Layer

The Service Layer adds a publish-find-bind model to the OSGi framework. A service in OSGi is a Java object implementing an interface that is published in a service registry under the name of the interface. Bundles may request services from the registry using interface names, to access the object implementing the interface. Bundles may also be notified when the registration state of the desired or observed interface changes in the registry.

Services in OSGi are published to the service registry by using the method *registerService(String, Object, Dictionary)* of the *BundleContext* object, which is provided to each bundle via the activator. The method takes an interface name, the object to register and a dictionary as parameters, and ensures that the object is actually an instance of the named interface. The dictionary represents properties of the service as key-value pairs. These properties can be used by bundles searching for services for filtering the results, by e.g. searching all services registered under the interface name *org.example.services.TransferService* that have the value "ACME" for the property key *vendor.id*. The method returns a *ServiceRegistration* object, which can be used to unregister the service with the corresponding method.

For accessing a service object of the registry, a bundle has to receive a *ServiceReference* from the OSGi runtime first. The *ServiceReference* contains the meta data of the service including its properties. To obtain a *ServiceReference*, the bundle needs to call the operation `getServiceReference(String)` or *getServiceReferences(String, String)*. The first returns a single *ServiceReference* of the desired interface name given as parameter. If more than one service object exists, the framework returns the service with the highest value in the property *service.ranking* or, if that does not lead to one result, the service that was registered first. The latter operation returns an array of all *ServiceReferences* to services implementing the desired interface. The second parameter is a string for filtering the results. The syntax of the filter string is based on the Lightweight Directory Access Protocol (LDAP) search filters, which are described in [How96]. For receiving the actual service object of a *ServiceReference*, the bundle has to call the method *getService(ServiceReference)* of the *BundleContext*. Alternatively to the simple service objects, a service factory can be published in the service registry. A service factory is an object of a class implementing the *ServiceFactory* interface. A service factory will return a new service object to each distinct requesting bundle.

Besides registering and receiving service objects programmatically, the service's provision and requirement can also be stated declaratively. The OSGi Service Compendium defines Declarative Services for this intention. As Declarative Services use the functionality of the Service Layer to register and bind the declared services, they do not extend the semantics of services and will thus not be covered here.

3.4 Security Layer

The Security Layer provides the security concept for OSGi bundles and is based on the Java 2 security model. Security configurations can be used to constrain the execution permissions of bundles on class level. As security aspects are not in the scope of this thesis, details on the Security Layer will be omitted here.

3.5 Comparison of Formal Component Specifications and OSGi

After inspecting the OSGi framework, its component model will now be compared to the component models examined in chapter 2. The areas of model features identified for the formal component specifications will be related to the component model of OSGi.

3.5.1 Provided Interfaces

The bundles are the elements of modularisation in OSGi. Bundles may provide functionality to other bundles. The package export mechanism provides means to hide implementation internals, as was proposed by Parnas in [Par72]. Services add another mechanism for information hiding by offering an interface name and a reference to the corresponding service object. In this case, functionality can be provided without the need to reveal internal implementation details. However, the content of the interface is not shared in the registry, but only the interface name – which is not necessarily unique – and some service properties.

3.5.2 Dependencies

Dependencies between bundles can be specified in three ways with OSGi:

- By referencing the symbolic name of the required bundle, all exported packages of a bundle are available in the referencing bundle.

- When required packages are referenced, the framework automatically resolves the dependencies, and accesses the packages, if a provider is available.
- When a required service is referenced with its interface name, the service is requested from the service registry.

All references are resolved at run time when the bundles are started, and errors will occur when dependencies cannot be satisfied.

For utilizing data types that are used by a service, each bundle referencing this service has to define a reference to the bundle providing that data type. This tight integration contradicts the specifications of formal component models.

For using a service object, the interface describing the service must be available to both, the providing and the requiring bundle. Besides keeping a copy of the interface in both, the consumer bundle (which requires the service) and the provider bundle (which provides the service), two approaches exist for making the interface available to both bundles. The approaches are depicted in figure 3.3. At the left hand side of the figure the interface is contained in the provider bundle. The consumer bundle has a dependency to the provider bundle to access the interface definition and the corresponding types. At the right hand side of the figure, a shared bundle contains the interface and types. The provider and the consumer need a dependency to the shared bundle to access the interface definition, thus the consumer and the provider do not need to know each other. This approach has the advantage that the consumer bundle can be developed without knowledge about the service provider, and thus without a static dependency. However, the service interface still needs to be defined before the consumer can be developed. Changes in the interface will require a recompilation of all consumers.

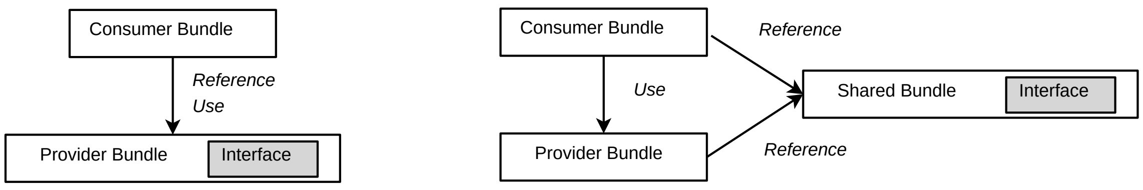


Figure 3.3: At the left hand side, the interface of the provided service is included in the provider bundle. The bundle consuming the service thus has a direct dependency on the provider bundle. At the right hand side, a shared bundle contains the service interface. The provider and the consumer bundle have dependencies to the shared bundle, thus the consumer can use the service without knowledge of the bundle implementing the service. [MBG10]

Formal component models address this feature at different levels of detail. In Palladio, SOFA 2, and UML the components communicate using shared interfaces. The interfaces are not part of any component, but separate entities. The problem with shared types is not addressed by these languages. In Π each component is completely context-independent. All imported and exported types and behaviour are defined locally in the components. I.e., the components have no static references to their context. UniCon addresses the problem similar to Π. In both languages, the connectors handle the conversion between required provided services and types. UniCon relies on predefined communication mechanisms in this case, while in Π conversion is defined by the developer.

3.5.3 Composition

While in formal component specifications composition is one of the key concepts, the concept cannot be found in the OSGi component model. Bundles are interconnected within a flat structure. No delegation of provisions or requirements to a composite can be defined.

3.5.4 Connectors

In contrast to most of the examined formal component specifications, OSGi does not define connectors as first class entities with possibly functional aspects. Thus complex interaction mechanisms like secure connections can hardly be considered. In OSGi, due to package access, several interaction mechanisms like e.g. shared memory can be used programmatically. The component concept of OSGi is focused on method invocation.

3.5.5 Communication Constraints

Most of the formal component specifications consider communication constraints in some way. In Π, for example, the permitted interaction of an exported type is defined with path expressions, preconditions, as well as usage requirements for imported types. The component model in OSGi does not consider interaction constraints in such detail. While the Security Layer allows for communication constraints to be defined, the constraints do not refer to the service's state but to method call origins. These constraints are given statically and have to be checked programmatically. Thus it is impossible to define permitted call sequences or concurrency constraints for method calls but programmatically.

3.5.6 Instantiation

Component instances can be called a key concept for reuse in SOFA 2, Π, UML, and UniCon. In OSGi, components must be distinct regarding the bundles' symbolic name and version. The installed bundles do not have instance names, and a bundle with the same symbolic name and version cannot be installed twice at the same time. For adding a second instance of a bundle, the bundle would have to be copied and the symbolic name or the version number would have to be changed. Although with this approach several instances of a bundle could be deployed, this could be considered confusing.

3.5.7 Assembly

Each examined formal component specification provides some sort of assembly. In Palladio a separate diagram is created for the assembly, and for the most languages an assembly is a special composite component, in which the interconnections can be defined manually.

As OSGi does not support component composition, this approach cannot be valid for the OSGi component model. A separate assembly time does not exist in OSGi. The bundles and services are interconnected automatically by the framework at run time instead. The connections cannot be explicitly defined, but just influenced by required and provided properties in the bundles' and services' meta data.

3.5.8 Quality Attributes

Some formal component specifications consider quality requirements or even provide tools for system simulation and analysis. In Palladio e.g. the components and other entities can be attributed with resource demands, and a simulation system is created in a deployment diagram. With this setup, an architecture can be simulated and analysed before it is implemented. Since the OSGi component model does not provide attributes for quality requirements, such functionality is not available.

3.5.9 Comparison Summary

The comparison between the core features of formal component models and the component model of OSGi shows great differences between the concepts. In formal component models, loosely coupled, self-describing, and hierarchically structured components communicate using connectors. In OSGi separate bundles are based on each other, each extending or using the functionality of another, while being tightly coupled to its dependency. Based on this information, a change to the OSGi Service Platform will now be proposed that aims at representing the concepts of formal component models in the practice-driven framework.

4 A Formal Component Model for OSGi

As shown in chapter 3, the OSGi component model strongly differs from the formal component specifications discussed in chapter 2. The OSGi component model lacks some essential features of formal component models. The main issues of components in OSGi is the tight coupling and the lack of concept for composite components. In this chapter, a new component model for OSGi will be proposed. The proposed model aims at implementing the features of formal component specifications in the practice-driven OSGi Service Platform. These features are implemented in three concepts of the proposed model: The provided interfaces and dependencies will be introduced in the component specification. These interfaces are annotated with behavioural specifications, representing the feature for communication constraints. The concept of connectors is represented as a first class entity in the model. A system assembly in the proposed model is a composite component. Composite components consist of instantiated subcomponents and their interconnection. The feature of providing information about quality attributes of an architecture is not addressed by the proposed model.

The model's concepts are specified first in section 4.1, before the reference implementation is described in section 4.2. Finally, the chapter proceeds by describing the tool support for the model's implementation in section 4.3.

4.1 Model Concepts

In this section the concepts of the proposed model are presented. The first class entities in the model are components and connectors. Components may be primitive and composite, introducing hierarchical architectures in the OSGi Service Platform.

4.1.1 Components

Components are first class entities in the proposed model. They have a name and interfaces for interconnection with their context. The interface can also have a behavioural description.

Primitive Components and Interfaces

Primitive components are bundles within the OSGi Service Platform with additional meta data. They have a name and provide or require services and types. Services are interfaces that are implemented by a single class within the providing component's bundle. Instances of this class are provided to the component's context. These services may use complex data types as parameters or return types that are not available in the platform, and which have to be provided to the context as well. Types are also represented by interfaces. In contrast to services, they are not necessarily implemented by a single class, but may have multiple implementations within one component.

Services and types are represented by Java interfaces within the providing or requiring bundle. A component configuration file is used to declare component meta data. For primitive components, these meta data are declarations of provided, required, and common parameter interfaces and their attributes.

The realization of a provided service is a class implementing the corresponding Java interface. Provided services may be singletons. I.e., each time the context requests an instance of the service, the same instance is returned, regardless of the requests origin. If the service is not declared a singleton, each request will be answered with a new instance of the service. For generating a new instance of the service, the default constructor is used. As some classes do not allow instantiation with this constructor, an instantiation method can be stated, which will be called instead. The instantiation method thus has to be a static method and return an instance of the provided interface. The descriptors of provided types just contain the type's interface.

The required services and types of a component are also Java interfaces within the bundle. The interface is declared to be required in the component configuration file. Required services are not implemented by a class within the bundle. Objects of classes implementing a required interface are bound to a binding class instead, which acts as a container for the required services. The required service declaration thus contains a binding class and a method to be called for binding. The binding method will be invoked with an instance of the required service as parameter. An unbinding method of the binding class can also be stated, which will be invoked when a required service should be removed. Likewise the classes implementing a provided service, the binding classes may be singletons and an instantiation method may be declared. The declarations of required types do not contain a binding class and a binding method, for they are not bound by the runtime. As they are instantiated within the functionality of the component instead of the framework, they also do not define an instantiation method and the singleton attribute.

The third category of component interfaces are common parameters. The common parameter semantics is based on the concept in Π. A common parameter is a required service or type that is also provided within one component. Figure 4.1 visualizes the concept. A component requiring a specific interface may provide the requirement to its context. The component is then called to be parameterized with this required interface. Different instances of the component may have different common parameters, and thus describe different functionalities. A common parameter service is bound to a component like a required service. Additionally to the attributes of required services, a method of the binding class must be defined that returns the instance of the common parameter service. An instance of the common parameter is bound to a binding class. As the same instance has to be returned on an invocation of the method providing the common parameter service, the binding class for a common parameter is always defined to be a singleton.

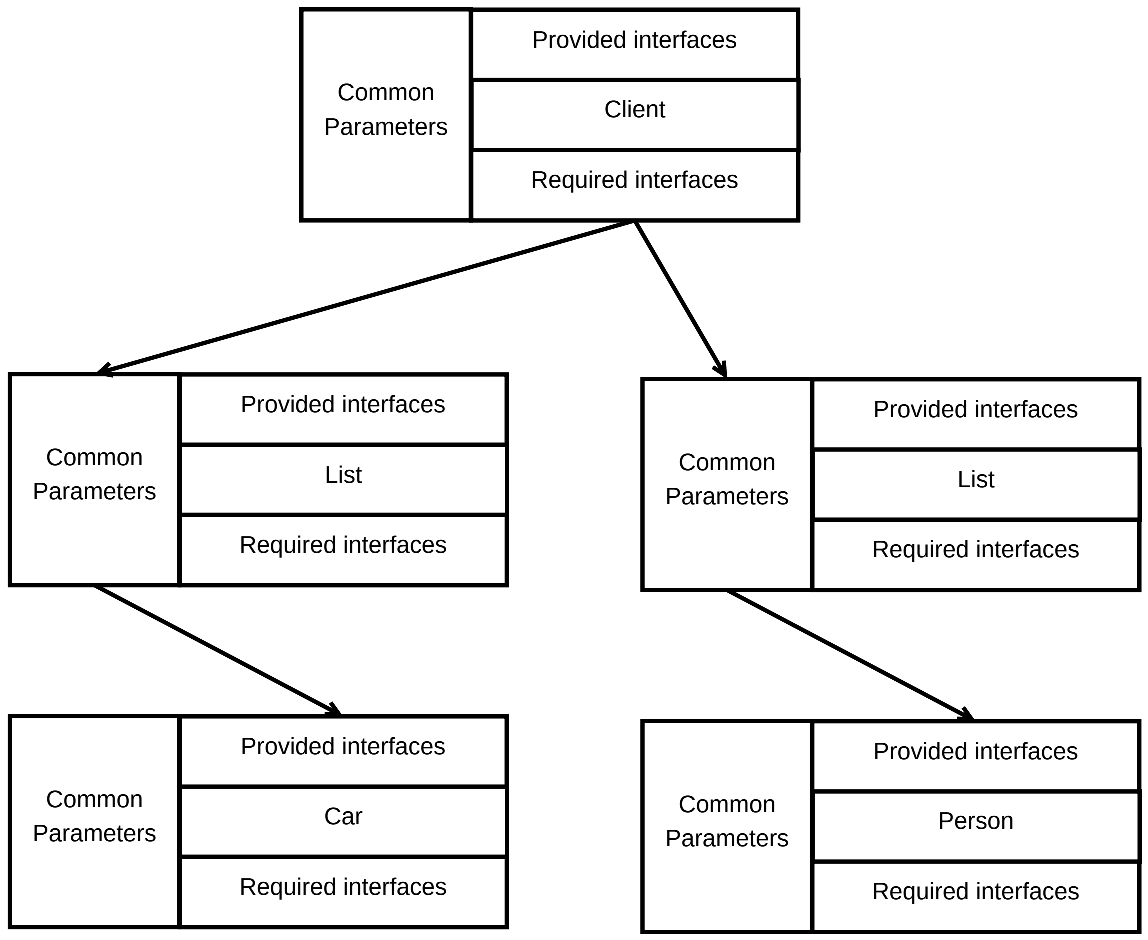


Figure 4.1: The client component requires two list services. Each list service uses another provided interface as a common parameter. The component *List* at the left side uses the interfaces provided by the component *Car*. The component *List* at the right side uses the interfaces of the component *Person*. The client has access to two lists, a list of cars and a list of people.

The provided and required services and types are defined locally in the component's bundle. The component is thus an independently compilable and deployable unit, because no dependencies to other components or the system are introduced at class level.

Listing 4.1 shows an exemplary component configuration file. The configuration file defines a set of provided and required services and types, as well as common parameters.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<msc:component xmlns:msc="msc.proposal.system.entities">

    <provides
        interface="org.example.IAuthentication"
        implClass="org.example.impl.AuthenticationImpl"
        singleton="true" />

    <provides type="org.example.IUser" />

    <requires
        interface="org.example.ILogin"
        bindClass="org.example.impl.Client"
        bindMethod="bindLogin"
        singleton="true" />

    <requires type="org.example.ISession" />

    <commonparameter
        interface="org.example.ILogin"
        bindClass="org.example.impl.Client"
        bindMethod="bindLogin"
        providesmethod="getLogin"
        instantiationMethod="getInstance" />

    <commonparameter type="org.example.ISession" />

</msc:component>
```

Listing 4.1: Component specification

Behavioural Specification for Interfaces

Additionally to the attributes stated in chapter 4.1.1, all component interfaces may also define a behavioural specification. The proposed component model uses interface automata [dAH01] for describing permitted behaviour.

Interface Automata An interface automaton is essentially a finite state machine with in- and output actions, where each input defines a received method call and each output defines an outgoing method call. This mechanism is used to describe how a component implementing this interface can be called by its context and how it makes calls to external components. Two interface automata interact by synchronizing their behaviour with input and output actions. I.e. when two automata share the same action names, the input and output steps with the shared action name can only fire synchronously. Interface automata use an *optimistic* approach to composition. I.e., two components are compatible if their respective interface automata share at least one possible path.

An interface automaton P is a six-tuple $P = \langle V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, T_P \rangle$.

- V_P is a set of states.
- $V_P^{init} \subseteq V_P$ is a set of initial states, with at most one state. P is called *empty* if $V_P^{init} = \emptyset$.
- A_P^I, A_P^O and A_P^H are disjoint sets of input, output and internal (hidden) actions. $A_P = A_P^I \cap A_P^O \cap A_P^H$ is the set of all actions.
- $T_P \subseteq V_P \times A_P \times V_P$ is a set of steps, which move the automaton from one state to another when the action is performed.

As an example, figure 4.2 shows the graphical representation of an interface automaton *User*. This interface automaton defines two input actions *ok* and *fail*, as well as one output action *msg*. The automaton consists of two states 0 and 1, of which 0 is the initial state. When the method *msg* is called by the component behind this interface, the automaton moves to the state 1. In this state the automaton just accepts a method call to *ok*. When this method is called by an external component, the automaton is in state 0.

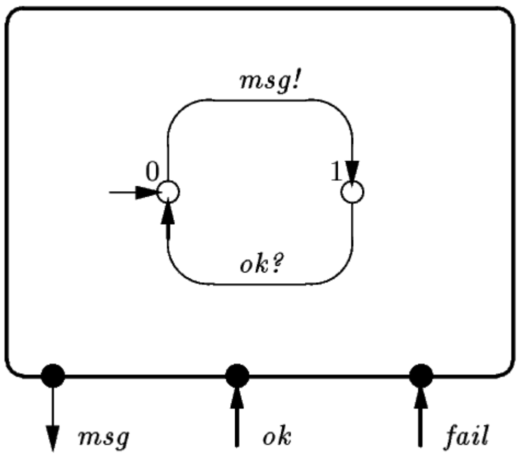


Figure 4.2: The interface automaton *User* has two states, 0 and 1, with 0 being the initial state. The transitions are annotated with the outgoing method call *msg* and the incoming method call *ok*. The incoming action *fail* is defined but can never be handled by the automaton. The context of the component has to make sure this method will never be called. [dAH01]

Two interface automata can be composable and compatible. The composition of two interface automata can be computed with algorithms explained in [dAH01]. The compatibility of interface automata is defined in [dAH01]:

> “Two interface automata P and Q are compatible *iff* (a) they are composable and (b) their composition is nonempty.”

Two interface automata are composable when they don’t share input or output actions, and their internal actions do not share their name with any action of the opposite interface automaton. This is necessary, because two interface automata synchronize on actions with similar names. For calculating the composition of two interface automata, the product has to be calculated first. The product of two automata $P \otimes Q$ is defined as follows:

- $V_{P\otimes Q} = V_P \times V_Q$
- $V_{P\otimes Q}^{init} = V_P^{init} \times V_Q^{init}$
- $V_{P\otimes Q}^{I} = (A_P^I \cup A_Q^I) \setminus shared(P,Q)$
- $V_{P\otimes Q}^{O} = (A_P^O \cup A_Q^O) \setminus shared(P,Q)$
- $V_{P\otimes Q}^{H} = A_P^H \cup A_Q^H \cup shared(P,Q)$
- $shared(P,Q) = (A_P^I \cap A_Q^O) \cup (A_P^O \cap A_Q^I)$ are the actions shared by the interface automata

$T_{P\otimes Q}$ is defined as

$$\begin{aligned} T_{P\otimes Q} =&\{((v,u),a,(v',u))|(v,a,v')\} \in T_P \wedge a \notin shared(P,Q) \wedge u \in V_Q\} \\ &\cup\{((v,u),a,(v,u'))|(u,a,u')\} \in T_Q \wedge a \notin shared(P,Q) \wedge u \in V_P\} \\ \cup\{((v,u),a,&(v',u'))|(v,a,v') \in T_P \wedge (u,a,u') \in T_Q \wedge a \in shared(P,Q) \wedge u \in V_P\}. \end{aligned}$$

If two interface automata are compatible, they have at least one shared path, which is defined by the composition. The composition of two interface automata is a *closed* operation, i.e. the composition of interface automata is also an interface automaton. Thus arbitrary numbers of interface automata can be composed. The composition of interface automata is also transitive, so the order of composing the automata is irrelevant.

Interface Automata in the Proposed Component Model The interface automata approach assumes shared interfaces which are used as required and provided interfaces. The proposed component model uses formal required as well as provided interfaces instead, which are interconnected with connectors. Thus the involved components do not necessarily use the same method names for communication. For this reason the interface automata approach was adapted to this situation as follows.

Provided interfaces only accept method calls. They are not used to make method calls to external components, and have no output actions. Thus A_P^O is empty. The actions are annotated with the signatures of methods in the interface. An example of an interface automaton for a provided interface is shown in figure 4.3. The interface has four methods, *init()* and *deinit()*, *getName()*, and *getLocation()*. The method *init* has to be called before any other method. The methods *getName()* and *getLocation()* may then be called repeatedly, before the method *deinit* may be called.

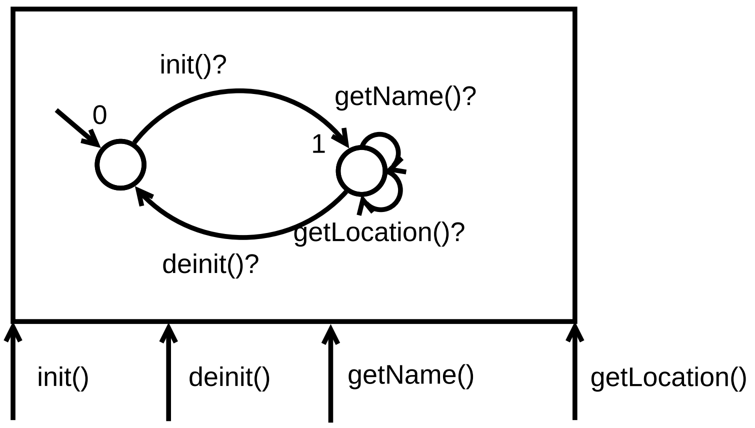


Figure 4.3: Interface automaton for a provided interface

Required interfaces just define outgoing method calls instead. They are not called by external components. Thus A_P^I is empty. Output actions are also annotated with method signatures. The interface automaton of a required interface is shown in figure 4.4. The interface has four methods, *create()*, *destroy()*, *getPOIName()*, and *getGeoLocation()*. The interface is to be used similarly to the interface in figure 4.3.

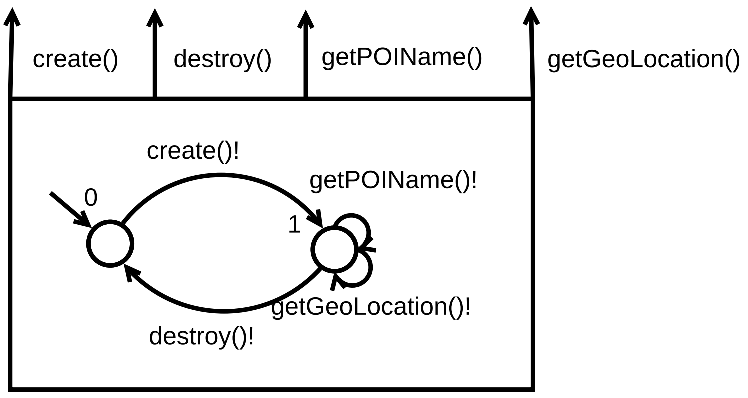


Figure 4.4: Interface automaton for a required interface

Interface automata for common parameter interfaces forward accepted method calls to another component. A common parameter interface needs an interface automaton with input actions, but without output actions, for checking the compatibility of a provided interface with the common parameter interface. For checking the compatibility of the common parameter interface with a required interface, an automaton with output actions is needed. Thus a common parameter interface needs to provide two interface automata. These interface automata just differ in the definition of actions as input or output actions. In the example in figure 4.5 the common parameter interface has four methods which are related to the methods in the required and provided interfaces described in the examples above.

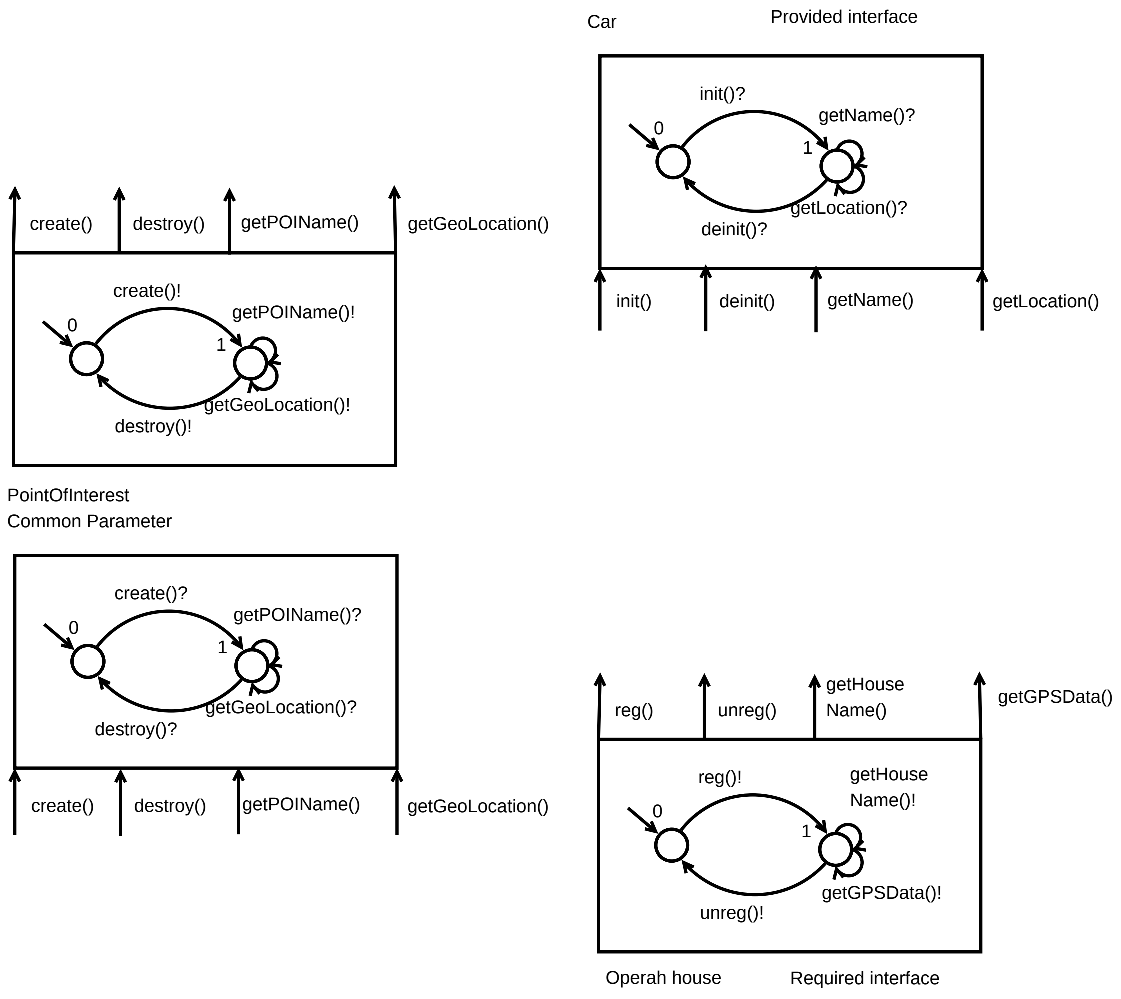


Figure 4.5: On the right side interface automata for a required and a provided interface are shown. The connection in this example is defined over a common parameter of a third component, which is shown on the left side. For checking the compatibility, the common parameter has to provide two interface automata which are highly similar. One defines the actions as input actions, and one defines the actions as output actions.

The original approach of interface automata requires action names to match for synchronisation of automata. This is expressed in the definition for shared actions $shared(P, Q)$ and for the actions of a product. As the proposed component model uses formal required and provided interfaces, these names do not necessarily match. For this reason the interface automata must be adapted for the product operation. A connector definition is needed that maps the methods of one interface to the methods of the other interface, and the actions of the involved automata have to be redefined. The connectors described in section 4.1.2 provide the necessary information. Instead of computing the product of two interface automata P and Q, the automata P' and Q' are used. The action names (i.e. the method signatures) of P' and Q' are concatenated. The new action names represent the method mapping defined in the connector.

Figure 4.6 shows an example, how the automata are adapted. The interface automata are mapped with the connector information. The resulting adapted automata are used for computing the product. Since only the input variables of the interface automata have been changed, all algorithms for computing the composition of interface automata can be reused.

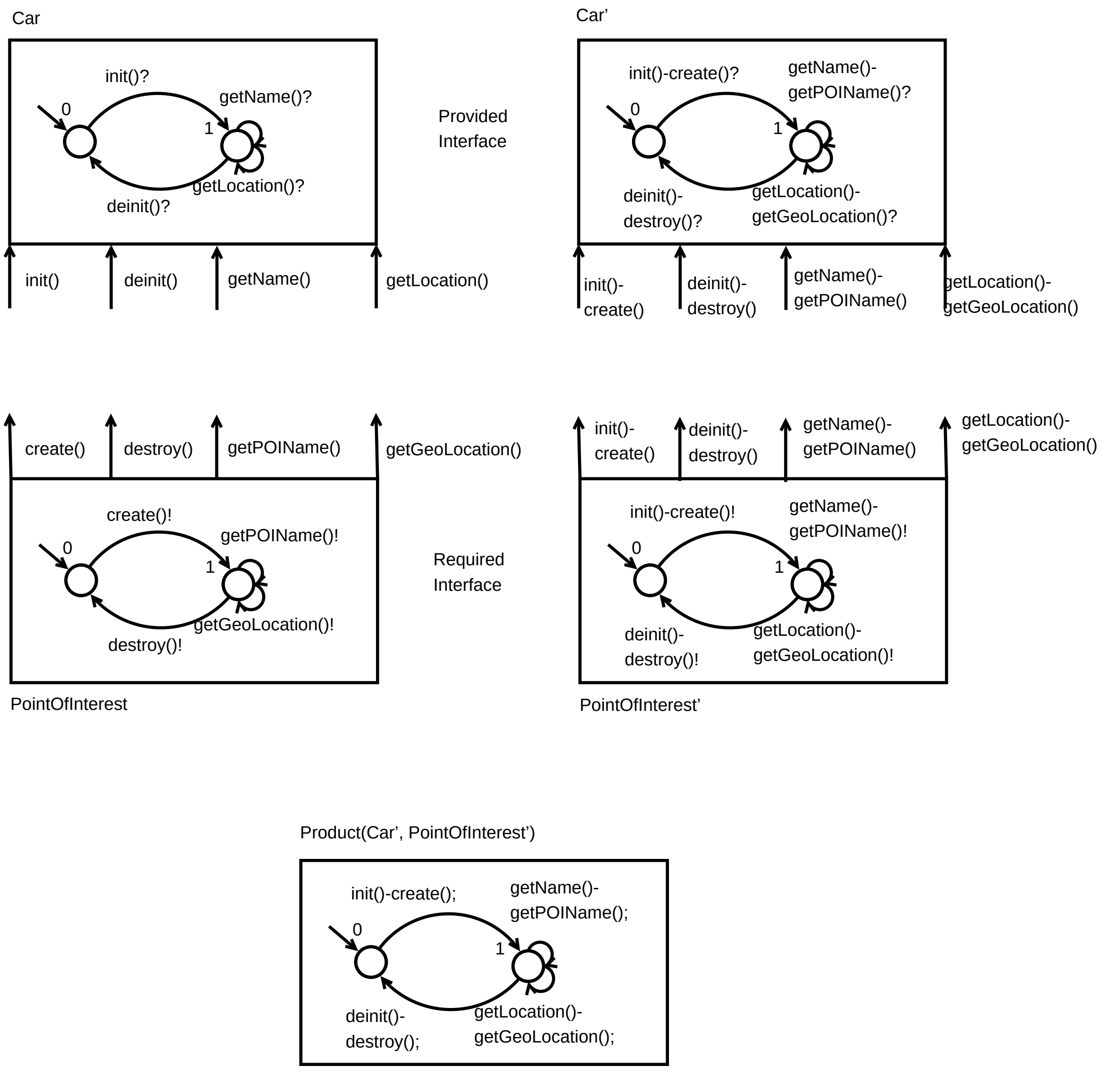


Figure 4.6: The upper and the lower left interface automata show the original automata P and Q. the upper and the lower right automata P' and Q' are adapted with connector information. The bottom interface automata is the product $P' \otimes Q'$.

Component Model Integration In the proposed component model, component interfaces may be attributed with a behavioural specification in terms of interface automata at development time. When the components are interconnected at assembly time, the interface automata's compliance can be checked manually, to find compatible interfaces and components. The correctness of the implementation regarding the automata is not monitored at run time.

Interface automata are defined in an own file for each interface declaration in the component descriptor. Listing 4.2 shows how to add the definition of an interface automaton to a provided interface.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<msc:component xmlns:msc="msc.proposal.system.entities">
    <provides interface="user.A" implClass="user.AImpl"
        behaviouralSpecification="MSC-OPT/A.ia.xml"/>
</msc:component>
```

Listing 4.2: Declaration of a behavioural specification in the component descriptor

The file `MSC-OPT/A.ia.xml` defines the interface automaton and is shown in listing 4.3. The automaton consists of three states, 0, 1, and 2 of which 0 is the initial state. Three output actions *init()*, *add(java.lang.String)* and *deinit()* reflect methods of the corresponding interface. The steps define that the method *init()* has to be called first, before the method *add(java.lang.String)* may be called. After *add*, a call to the method *deinit()* is expected, before the automaton is in state 0 again.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:interfaceautomata xmlns:ns2="msc.proposal.system.interfaceautomata.
    entities">
    <states name="0" initial="true" />
    <states name="1" initial="false" />
    <states name="2" initial="false" />

    <actions type="output" name="init()" />
    <actions type="output" name="add(java.lang.String)" />
    <actions type="output" name="deinit()" />

    <steps to="1" from="0" action="init()" />
    <steps to="2" from="1" action="add(java.lang.String)" />
    <steps to="0" from="2" action="deinit()" />
</ns2:interfaceautomata>
```

Listing 4.3: An interface automaton defined in XML

The definition of common parameters requires an automaton for the provided and the required view. An example of a behavioural description for common parameters is shown in listing 4.4.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<msc:component xmlns:msc="msc.proposal.system.entities">
    <provides interface="user.CP" implClass="user.CPImpl"
        behaviouralSpecificationProviding="MSC-OPT/CPProv.ia.xml"
```

```
      behaviouralSpecificationRequiring="MSC-OPT/CPReq.ia.xml"/>
</msc:component>
```

Listing 4.4: Declaration of behavioural specifications for a common parameter

4.1.2 Connectors

As described above, components define their required and provided interfaces locally. No shared interfaces can be used between two components, as the interface one component uses is not known to the other component. Hence the interfaces might have different names and method definitions. For interconnecting a required with a provided interface, these interfaces and their methods need to be mapped. For the required and provided services to be bound at assembly time, a connector must be introduced.

The connectors are first class entities in the proposed component model. For mapping interfaces and methods, information about required and provided interfaces of the components to be interconnected and their methods is needed. When a method of a required service or type is invoked, the connector maps the call to an instance of the provided counterpart. The mapping is transparent to the involved components. The mapping of formally described component interfaces and their methods allow for context-independent components. The components do not have to share interfaces for communication, but define their requirements locally. Connectors can be extended to have arbitrary functionality. An extension of the standard connector has been implemented and is described in chapter 4.2.2.

The mapping information for connectors is given as key-value pairs in a properties file. In the exemplary mapping file shown in listing 4.5, the required service *org.example.ILogin* is mapped to the service *org.example.IAuthentication*. The types are mapped accordingly. The remaining rows define method mappings between the services and types.

```
org.example.ILogin=org.example.Authentication
org.example.ILogin.login(java.lang.String,java.lang.String)=org.example.
   Authentication.auth(java.lang.String,java.lang.String)
org.example.ILogin.getSessionData()=org.example.Authentication.getUser()

org.example.ISession=org.example.IUser
org.example.ISession.getSessionName()=org.example.IUser.getSessionName()
org.example.ISession.getId()=org.example.IUser.getUsedId()
org.example.ISession.setLastLogin(java.lang.Date)=org.example.IUser.
   addLastLogin(java.lang.Date)
```

Listing 4.5: Example of a service and type mapping between two components

4.1.3 Composition and Assembly

Hierarchical component architectures in the proposed model are enabled by composite components. Primitive components implement their functionality with Java code, while composite components implement their functionality with the instantiation and binding of subcomponents. A subcomponent is referenced by the composite with a location and an instance name. They are then interconnected with connectors. A delegate can be used for the composite component to provide or require the services and types that a subcomponent provides or requires. Delegates are special types of connectors, which bind required interfaces of a subcomponent to required interfaces of the composite or provided interfaces of a subcomponent to provided interfaces of the composite. The mapping information for the connectors and delegates must be available within the composite component.

The subcomponents are also bundles in the OSGi Service Platform and can thus be accessed by all other bundles, regardless of an intended component hierarchy. I.e., composite components do not completely hide their implementation details, as this would mean to hide the existence of the subcomponents. While the direct access to subcomponents from other components is possible, such a setup would break the hierarchical composition of components.

A system assembly of a complete software system in the proposed component model can be achieved in two ways. First, components may be installed and interconnected manually. The manual binding is described in chapter 4.2.2. Second, components can be declared the subcomponents of a composite component, which contains the bindings and is deployed as a whole.

The example in listing 4.6 shows the configuration file of a composite component. The composite component has two subcomponents, *myClient* and *myDir*. Both subcomponents have a relative location, i.e. the files representing the subcomponents are deployed within the composite component. The location attribute accepts any bundle location that Equinox accepts. In this example, a connector between the component *myClient* and the component *myDir* is defined, as well as a delegate connecting the provided interfaces of *myClient* to the composite.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<msc:component xmlns:msc="msc.proposal.system.entities">
    <subcomponent instanceName="myClient" location="./Client_1.0.0.jar" />
    <subcomponent instanceName="myDir" location="./UserDir_1.0.0.jar" />

    <connector required="myClient" provided="myDir" bindFile="MSC-OPT/
        innerBinding" />
    <delegate provided="myClient" bindFile="MSC-OPT/delegations" />
</msc:component>
```

Listing 4.6: Example of a service and type mapping between two components

4.2 Implementation

A reference implementation of the proposed component model for OSGi was developed in the Equinox framework during this thesis. The Eclipse Equinox framework [Ecl10c] is a wide-spread open source implementation of the OSGi specification. It is mainly used as the runtime platform for the Eclipse Integrated Development Environment (IDE) [Ecl10b].

4.2.1 Components and Interfaces

Primitive components in the proposed model are OSGi bundles with additional meta data. This meta data consists of required, provided and common parameter interfaces, as well as a behavioural specification, and an instance name. For representing this information at run time, the interface *Bundle* of the OSGi reference implementation was adapted. Different methods for obtaining the instance name, the names of the component interfaces, and the behavioural description were added to this interface. The implementation of this interface in Equinox is the abstract class *AbstractBundle*, which is the foundation for a set of bundle types. This class was adapted to read the component descriptors when the bundle is initialized. The data read from the configuration file is stored in the bundle object, to serve the methods that request the component interface definitions and the behavioural description. Additionally, the constructor and its invocations were adapted to provide an instance name to the bundle object.

For representing composite components, four methods controlling the life cycle of the component had to be adapted. The methods *start*, *stop*, *uninstall*, as well as the constructor of *AbstractBundle* have been customized to consider subcomponents. When a component configuration file defines subcomponents, the constructor of *AbstractBundle* initializes the subcomponents before finishing its execution. Thus, when a composite is installed, it autonomously installs its subcomponents before it finishes its own installation process. The same procedure was introduced in the methods *start*, *stop*, and *uninstall*.

Several methods were added to the interface *Bundle* due to the connectors. The method *getInstance(String)* returns an instance implementing the provided or common parameter interface, that is referenced by the parameter. With the method *getBindingInstance(String)*, an instance of a binding class can be obtained, that is defined to bind the required or common parameter interface named by the parameter. When the method *getInstance* is invoked on a primitive component, the component first tries tries to find and instantiate the class that implements the service defined by the parameter. If the instance is a common parameter, it is retrieved by calling the *providesMethod* on the binding class of the common parameter. Composite components have to use delegating connectors to get a service instance from a subcomponent. Delegates are described in chapter 4.2.3. The method *getBindingInstance* works analogously for binding instances.

The methods *bind(String, Object)* and *unbind(String, Object)* are used to bind and unbind objects that define a required or common parameter service. The first parameter is the name of the required service to be bound, while the second parameter is the object that implements that interface and thus represents the service. The method *unbind* needs to be called with the service object as parameter, to cover the case that a component does not store a single instance, but a list of service objects.

The command line interpreter of the Equinox command console was also extended. The install command now takes an instance name as optional parameter. When no instance name is given, the framework sets the instance name of the component to be installed to the symbolic name of the bundle.

Interface Automata

The interface automata are implemented by a class structure that directly represents the interface automaton's elements and their dependencies. A class *InterfaceAutomaton* contains references to sets of *Actions*, *Steps*, and *States*, of which one may be an initial state. A utility class *InterfaceAutomataUtils* implements all necessary computations described in [dAH01], by using Java's Collections API.

4.2.2 Connectors

A connector in the implementation of the proposed model is an instance of the *Connector* class. The connector class is responsible for binding provided service instances to requiring components and to perform the interface and method mapping.

Basic Connector

The class *Connector* implements three different connector types. The first type connects required interfaces of one component with the provided interfaces of another component. The second type connects the required interfaces of a subcomponent with the required interfaces of its composite. The third type connects the provided interfaces of a composite with the provided interfaces of a subcomponent. The last two connector types implement the delegate functionality. The implementation of delegates is explained in chapter 4.2.3.

A connector binds components in a specific direction. The component that is connected to another component is stored as the *fromBundle*, while the component it is connected to is stored as the *toBundle*. For performing the interface and method mapping, the connector also stores the mapping information retrieved from the properties file. When a connector is constructed, it requests the required, provided, and common parameter interfaces from the involved components, as well as the behavioural specification. The connector also automatically composes the interface automata of each mapped pair of interfaces, for later reference. However, the result of the automaton composition is currently not checked at run time. Finally, the connector has a state, indicating whether it is bound or not.

The actual binding of a connector is executed in the method *bind*. In this method, a connector first requests instances for all provided and mapped interfaces from the *toBundle* by using the method *getInstance* of the interface *Bundle*, which represents the component. The retrieved service instances are then wrapped in a dynamic proxy [Sun10] with the mapping information.

A dynamic proxy class is a class implementing one or more interfaces which is specified at run time, without the need for prior generation of the class. A proxy class can be transparently used as an instance of the specified interfaces. When a method is invoked on an instance of the proxy class, an *InvocationHandler* object handles the method invocation. The method call is dispatched to a method of the *InvocationHandler* called *invoke*. The parameters of this method are: the proxy object that dispatched the method call, a *Method* object reflecting the original method call, and the actual parameters of the original method call as parameters. The return value of the method is an *Object*. The object that this method returns is the return value of the original method invocation on the proxy object.

The concept of dynamic proxies is used in the reference implementation for wrapping the provided services and types. Each instance of a provided interface is wrapped with a dynamic proxy. The invocation handler of the proxy is an instance of the class *DelegatingInvocationHandler*. The connector creates an instance of this class with the provided service instance and the mapping as parameter. The proxy object is defined to implement the required interface. To actually bind the proxy object to the required service, the method *bind(String, Object)* is called on the *Bundle* object of the requiring component. The service binding is depicted in figure 4.7 for reference.

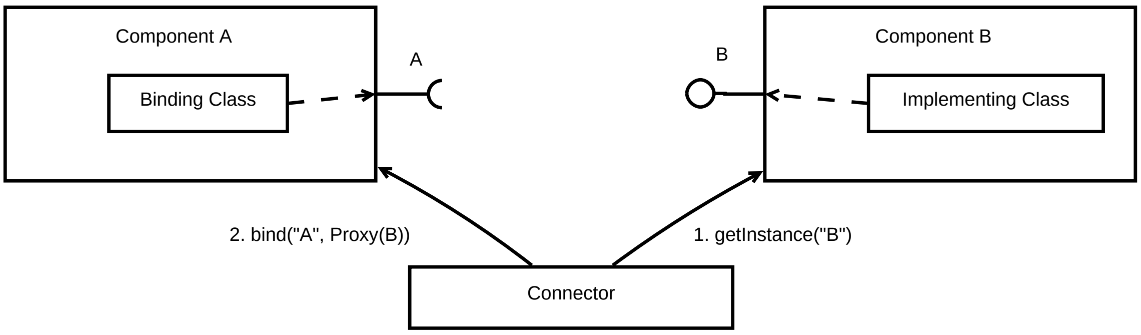


Figure 4.7: The connector requests a service instance from the providing component and binds it to the requiring component.

The *DelegatingInvocationHandler* is responsible for the method and type mapping. When a method is called on a required interface that is implemented by a proxy, the method *invoke* of the *DelegatingInvocationHandler* is called. The handler first checks whether the method is mapped. If the method is not mapped, the original method is invoked on the provided instance and the return value is returned to the caller. This is the case when a method of the type *Object* is called on a provided service or type, for example.

When the method is to be mapped by the connector, the handler first loads the mapped method from the provided interface class. Before actually calling the provided method, the parameters are checked for mapped types. If a parameter is of a mapped type, the object must be wrapped by a dynamic proxy with a new instance of the *DelegatingInvocationHandler*. As this parameter is already a proxy instance which will map the method calls or an object that is directly usable by the requiring component, it needs to be wrapped by another proxy instance that reverses the mapping, and thus allows the provider to access the parameter. After the provided method is invoked on the original service, the return type is checked for a mapped type. If the type is mapped, the return value is also wrapped with a proxy for the requiring component to use.

The user can establish a connection between two components in two ways: First, the components can be embedded into a composite, which defines a connector between its subcomponents. Second, the Equinox command line can be used to define a connection. The command *msc_bind* was implemented for this task. The command is to be called with the instance names of the components to be interconnected, and a location of a binding file as parameters.

RMI Connector

The Remote Method Invocation (RMI) [GJSB05] connector is an extension to the basic connector. RMI provides a Remote Procedure Call (RPC) mechanism to Java, allowing to invoke methods on objects which reside on another Java Virtual Machines (JVM), on possibly remote hosts. In RMI, a server publishes objects with a name in an RMI registry. This registry can be queried by RMI clients, to find objects under the given name. For using the object, a client needs to know the interface that the object implements. Additionally, this interface needs to fulfill certain requirements: It has to extend the interface *Remote* and each method signature needs to declare the exception type *RemoteException* to be thrown.

In the reference implementation of the proposed model, the RMI connector is separated into two main parts. The structure of a connection with the RMI connector is depicted in figure 4.8. The class *RMIProvider* is used to publish interfaces in a local registry. The class *RMIConnectorClient* is used to create a connection to the registry and to obtain the desired object.

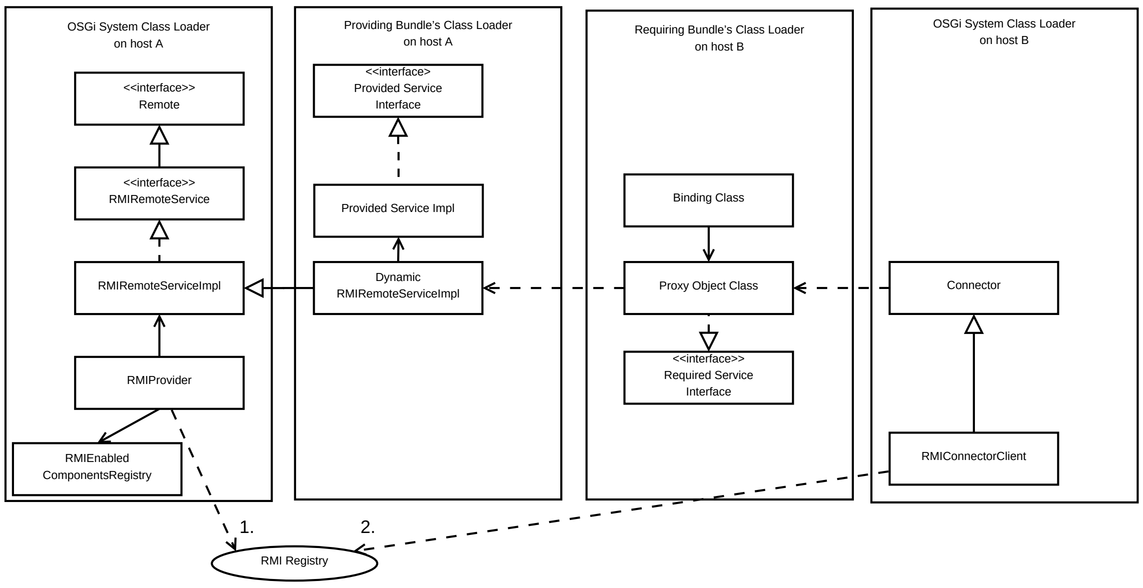

OSGi System Class Loader on host A
<<interface>> Remote
<<interface>> RMIRemoteService
RMIRemoteServiceImpl
RMIProvider
RMIEnabled ComponentsRegistry
Providing Bundle's Class Loader on host A
<<interface> Provided Service Interface
Provided Service Impl
Dynamic RMIRemoteServiceImpl
Requiring Bundle's Class Loader on host B
Binding Class
Proxy Object Class
<<interface>> Required Service Interface
OSGi System Class Loader on host B
Connector
RMIConnectorClient
1.
2.
RMI Registry

Figure 4.8: With the RMI connector, an *RMIProvider* publishes a provided service object to a RMI registry. The service object is received by the *RMIConnectorClient*, which binds the remote object to the binding class and performs the method mapping.

The services to be published are on the RMI server side. As interfaces of objects that are published in the RMI registry need to extend the interface *Remote*, the class *RMIProvider* cannot publish the provided interfaces of a component. A proxy object of the type *RMIRemoteServiceImpl* is published instead, which implements an interface with a method *invoke*, which can be published to an RMI registry. The parameters of this method are an object of the class *Method*, reflecting the method to be invoked, the parameter types and the actual parameters. With this information, the proxy object invokes the desired method on the service object.

OSGi uses a hierarchical model of class loading behaviour for bundles and the framework. Each bundle has its own class loader and may access the class loaders of the bundles providing its imported packages. As the framework does not import any package, it cannot access the class loaders of bundles. The *RMIRemoteServiceImpl*, which is implemented in the framework, can thus not directly call methods on the provided service implementation, because it cannot load the class of the object to be called. To resolve this issue, a subclass of the class *RMIRemoteServiceImpl* is generated dynamically at runtime using Javassist [Chi98], a library for generating classes and editing byte code at run time. The generated class is located in the class loader of the bundle, thus it may access the provided service class. The RMI enabled service and a list of components providing interfaces with RMI is then published in a RMI registry by the class *RMIProvider*.

The component requiring the remote service is on the client side of the RMI connection. The class *RMIConnectorClient* extends the functionality of the *Connector* by receiving component descriptors and service instances from an RMI registry. Just as the basic connector, the RMI connector wraps the received instances in dynamic proxies for performing the method mapping. As the RMI connector receives instances of the interface *RMIRemoteService* instead of service objects, the invocation handler has to dispatch the method calls to the method *invoke*. These method calls are then transported via RMI to the dynamically generated subclass of *RMIRemoteServiceImpl* on the server side, which finally invokes the desired method on the provided service.

In RMI, a method call with an object as parameter or return value that does not implement the interface *Remote* , results in a call by value. I.e., the object is serialized for the transport. Thus the provided implementation of a provided data type, that is to be connected via the RMI connector, must implement the interface *Serializable*. The usage of the RMI connector is thus not transparent to the providing component.

The Equinox console was extended by the command *msc_provide_rmi* for publishing a component's provided interfaces in a RMI registry. The parameters of this command are the instance name of a component and, optionally, the port of the local RMI registry. The command *msc_bind* accepts a RMI URL (e.g. rmi://localhost) as last parameter. If this parameter is given, the class *RMIConnectorClient* is used for the connection, instead of the class *Connector*.

4.2.3 Composite Components

As explained in section 4.2.2, a connector has one of three types. The first type is described in that section. The second type connects the required interfaces of a subcomponent with the required interfaces of its composite. The third type connects the provided interfaces of a composite with the provided interfaces of a subcomponent. Composite components are interconnected with their subcomponents with these delegating connectors.

A provided delegate stores the interface and method mapping to the subcomponent's provided interfaces. When the method *getInstance* is invoked on a composite component, it recursively requests its subcomponents for an instance of the interface. Before each subcomponent is requested, the delegating connector to the subcomponent is used to map the composite interface name to the subcomponent's interface name. When a subcomponent is found that provides an instance for the interface, the instance is wrapped in a proxy class object that directly maps the required interfaces and methods of the requesting component to the provided interfaces and methods of the subcomponent. Thus the mapping of the requiring component to the composite and from the composite to its subcomponent are summarized to create a smaller stack of proxy objects. The method *getBindingInstance* works analogously with binding classes.

The composite component's methods *bind* and *unbind* use the delegating connectors to find a subcomponent providing a binding class for the required service and dispatch the method invocation the the corresponding subcomponent after mapping the name of the required interface.

4.3 Tool Support

The elements to be defined for assembling an application in the proposed component model can be completely defined in XML or properties files. Hence no special tools are required for defining components and assemblies. However, tools could enhance the efficiency of working with the component model in both phases: at development time and at assembly time.

4.3.1 Development Time

At development time, primitive components are specified and implemented. In the proposed component model, the required, provided, and common parameter interfaces are defined in the component configuration file at development time. Additionally, the behavioural specification of interfaces may be given using interface automata.

Interface Definition

For defining interfaces within a component to be a provided, required or common parameter interfaces, they must be referenced in the component descriptor file as data type or service. This might be an error prone and complex task. In addition, service interfaces may have structural references to data type interfaces that also have to be considered. A tool could be used to identify interfaces and their dependencies within the component, for defining them as provided, required, or common parameters and to distinguish between service interfaces and data types.

Several heuristics could be used to identify those categories automatically. Interfaces that are used but not defined within a component may be required interfaces or common parameters, while interfaces that are implemented may be provided interfaces. Data types could be distinguished from service interfaces by searching for required or provided methods that take the interface of interest as parameter or use it as return value. Interfaces that are never used as parameter or return value might be service interfaces, while the others might be data types. While these are just heuristic criteria, an automatic identification of those categories for interfaces may help the architect to efficiently describe component interfaces with the proposed model.

Interface Automata Definition

Interface automata are defined manually in XML files. These files may become large and confusing when the automaton has many elements. A clear structuring of the document can help to clarify the structure of the automaton, but for a better overview of the automaton, a graphical tool could be helpful.

4.3.2 Assembly Time

At assembly time, components are interconnected in system assemblies and composite components. The main tasks at assembly time are to find components that provide the functionality needed by other components, and to define the interface and method mapping for connectors.

Component Matching and Mapping Generation

The formal import and formal export of components leads to the need for defining connectors that describe a mapping between the requiring and the providing component interfaces. Behavioural interfaces and types need to be mapped as well as their methods. In non-trivial systems these mappings may be very large and complex to be created. A tool helps to define these mappings by using a set of criteria for comparing interfaces and methods.

Several criteria may be used for comparing interfaces, and comparison has to take place on three layers:

1. Components can be compared by comparing their interfaces
2. Interfaces can be compared by comparing their methods
3. Methods can be compared by comparing their signatures

Criteria for comparing methods are the return and parameter type matching as well as the method name matching. The equality of method names are an indicator of a good matching. The matching of the parameter and return type is another indicator. If the parameter count and the parameter types of a method are the same, the methods might have the same semantics. For type comparison four cases must be considered:

1. The types of the method in the provided and the required interface are shared, e.g. *java.lang.String*. In this case the method types can be directly compared. The types declared by the providing interface might be more specialised in a type inheritance than those declared by the requiring interface.
2. The type is provided by the providing component and required by the requiring component. In this case the types can be directly compared to each other.
3. The type is required by both interfaces. In this case the type interfaces might be compared.
4. The type is provided by the required interface and required by the provided interface. In this case the types could be directly compared. This situation should be avoided, as the architecture is a cyclic graph.

In the cases 2 to 4 the comparison of the type interfaces may lead to a recursive call of comparing types if the type graph is a cyclic graph.

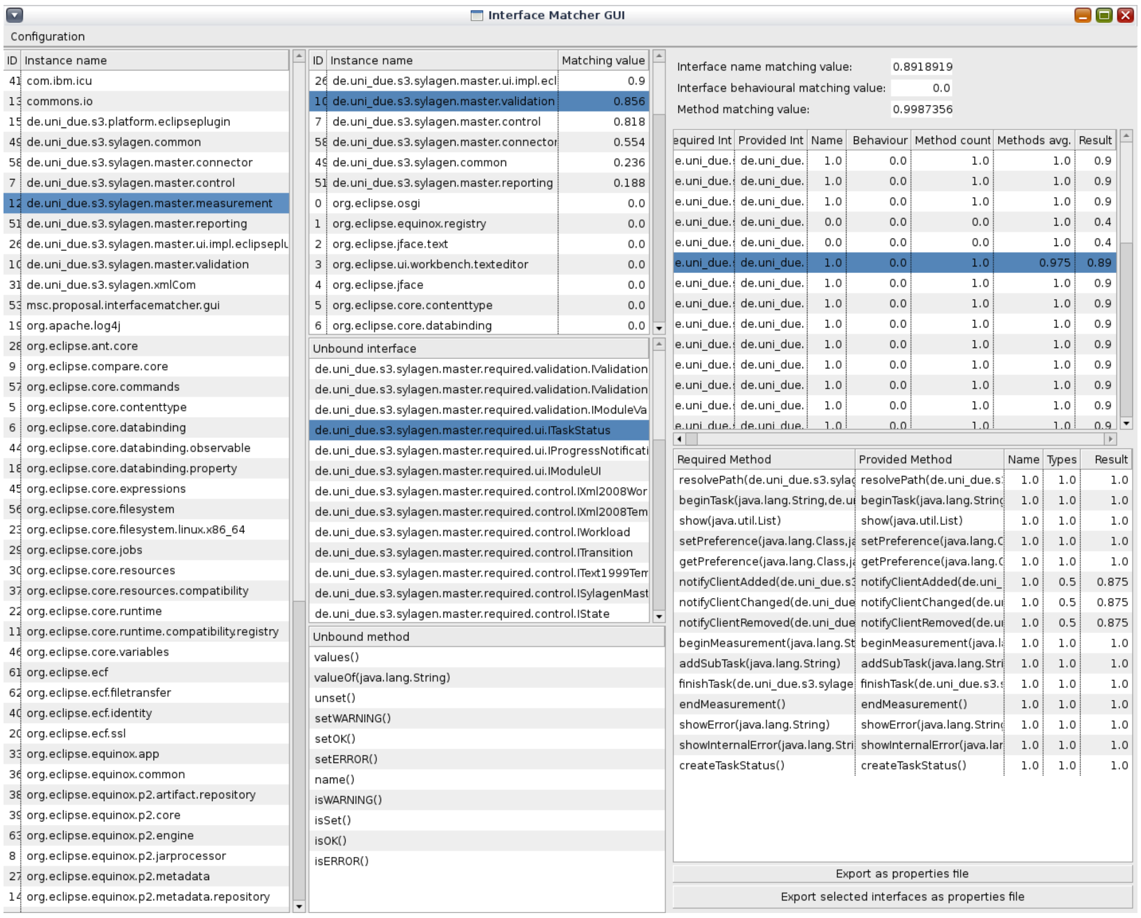
Criteria for matching interfaces are the interface name matching, the average of the method matching value for the best matching methods, the method count matching, and the behavioural compliance. The interface name might be an indicator for interface equality. In addition, the methods are indicators: the same number of methods in an interface provides information about a possible matching, as well as the average method matching value of the most matching methods, which is computed by the criteria described above.

The behavioural compliance in terms of interface automata can be defined by $1 - |A^I_{P\otimes Q} \cap A^O_{P\otimes Q}|/|A_{P\otimes Q}|$, i.e. the quotient of the number of external (input and output) actions and the total number of actions of the product automaton. The result ranges from 1 for two completely compatible automata to 0 for completely incompatible automata. With this information, two components can be compared by the matching values of their most matching interfaces: the average behavioural compliance, the average interface name matching and the average method matching can be summarized to one comparison value.

The comparison criteria can be divided into hard and soft criteria. The type comparison and the behavioural compliance are hard criteria that can be formally checked. The name matching is a soft criterion, i.e. the names can be checked for direct equality or for semantical similarity. As the comparison criteria are not validated empirically, other criteria may be of interest. The single criteria also need to be weighted.

An experimental tool has been implemented during this thesis to support the case study explained in chapter 5.1. The tool supports the architect at assembly time by matching components and generating mapping configuration files for connectors, using the presented concepts. The user interface of the tool is divided into three columns. In the left column, all bundles installed in the runtime are shown. If the user selects a bundle, it is interpreted as a requiring component against which all other installed bundles are matched as providing components. The matching result is shown in the upper table of the middle column. The lower tables in the middle column show the interfaces of the requiring component that are currently not bound at run time and their unbound methods. When a providing component is selected in addition to the requiring component, the right column shows the matching details, including the most matching interfaces and the most matching methods of these interfaces, as well as the corresponding matching results. Buttons on the lower right corner allow for generation of a connector mapping file of all or selected interfaces. The matching can be configured using the *Configuration* menu which allows to define weights for the single comparison criteria.

For the comparison of the interfaces, the matching program loads the component interfaces of the components using their respective class loaders. The content of the interfaces and their method signatures are examined using Java's reflection mechanisms.


Interface Matcher GUI
Configuration
Export as properties file
Export selected interfaces as properties file

ID	Instance name
41	com.ibm.icu
13	commons.io
15	de.uni_due.s3.platform.eclipseplugin
49	de.uni_due.s3.sylagen.common
58	de.uni_due.s3.sylagen.master.connector
7	de.uni_due.s3.sylagen.master.control
12	de.uni_due.s3.sylagen.master.measurement
51	de.uni_due.s3.sylagen.master.reporting
26	de.uni_due.s3.sylagen.master.ui.impl.eclipseplu
10	de.uni_due.s3.sylagen.master.validation
31	de.uni_due.s3.sylagen.xmlCom
53	msc.proposal.interfacematcher.gui
19	org.apache.log4j
28	org.eclipse.ant.core
9	org.eclipse.compare.core
57	org.eclipse.core.commands
5	org.eclipse.core.contenttype
6	org.eclipse.core.databinding
44	org.eclipse.core.databinding.observable
18	org.eclipse.core.databinding.property
45	org.eclipse.core.expressions
56	org.eclipse.core.filesystem
23	org.eclipse.core.filesystem.linux.x86_64
29	org.eclipse.core.jobs
30	org.eclipse.core.resources
37	org.eclipse.core.resources.compatibility
22	org.eclipse.core.runtime
11	org.eclipse.core.runtime.compatibility.registry
46	org.eclipse.core.variables
61	org.eclipse.ecf
62	org.eclipse.ecf.filetransfer
40	org.eclipse.ecf.identity
20	org.eclipse.ecf.ssl
33	org.eclipse.equinox.app
36	org.eclipse.equinox.common
38	org.eclipse.equinox.p2.artifact.repository
39	org.eclipse.equinox.p2.core
63	org.eclipse.equinox.p2.engine
8	org.eclipse.equinox.p2.jarprocessor
27	org.eclipse.equinox.p2.metadata
14	org.eclipse.equinox.p2.metadata.repository

ID	Instance name	Matching value
26	de.uni_due.s3.sylagen.master.ui.impl.ecl	0.9
10	de.uni_due.s3.sylagen.master.validation	0.856
7	de.uni_due.s3.sylagen.master.control	0.818
58	de.uni_due.s3.sylagen.master.connector	0.554
49	de.uni_due.s3.sylagen.common	0.236
51	de.uni_due.s3.sylagen.master.reporting	0.188
0	org.eclipse.osgi	0.0
1	org.eclipse.equinox.registry	0.0
2	org.eclipse.jface.text	0.0
3	org.eclipse.ui.workbench.texteditor	0.0
4	org.eclipse.jface	0.0
5	org.eclipse.core.contenttype	0.0
6	org.eclipse.core.databinding	0.0

Unbound interface
de.uni_due.s3.sylagen.master.required.validation.IValidation
de.uni_due.s3.sylagen.master.required.validation.IValidation
de.uni_due.s3.sylagen.master.required.validation.IModuleVa
de.uni_due.s3.sylagen.master.required.ui.ITaskStatus
de.uni_due.s3.sylagen.master.required.ui.IProgressNotificati
de.uni_due.s3.sylagen.master.required.ui.IModuleUI
de.uni_due.s3.sylagen.master.required.control.IXml2008Wor
de.uni_due.s3.sylagen.master.required.control.IXml2008Tem
de.uni_due.s3.sylagen.master.required.control.IWorkload
de.uni_due.s3.sylagen.master.required.control.ITransition
de.uni_due.s3.sylagen.master.required.control.IText1999Ter
de.uni_due.s3.sylagen.master.required.control.ISylagenMast
de.uni_due.s3.sylagen.master.required.control.IState

Unbound method
values()
valueOf(java.lang.String)
unset()
setWARNING()
setOK()
setERROR()
name()
isWARNING()
isSet()
isOK()
isERROR()

Interface name matching value:	0.8918919
Interface behavioural matching value:	0.0
Method matching value:	0.9987356

equired Int	Provided Int	Name	Behaviour	Method count	Methods avg.	Result
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	0.0	0.0	0.0	1.0	0.4
e.uni_due.	de.uni_due.	0.0	0.0	0.0	1.0	0.4
e.uni_due.	de.uni_due.	1.0	0.0	1.0	0.975	0.89
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9
e.uni_due.	de.uni_due.	1.0	0.0	1.0	1.0	0.9

Required Method	Provided Method	Name	Types	Result
resolvePath(de.uni_due.s3.sylag	resolvePath(de.uni_due.s	1.0	1.0	1.0
beginTask(java.lang.String,de.u	beginTask(java.lang.String	1.0	1.0	1.0
show(java.util.List)	show(java.util.List)	1.0	1.0	1.0
setPreference(java.lang.Class,ja	setPreference(java.lang.C	1.0	1.0	1.0
getPreference(java.lang.Class,ja	getPreference(java.lang.C	1.0	1.0	1.0
notifyClientAdded(de.uni_due.s3	notifyClientAdded(de.uni_	1.0	0.5	0.875
notifyClientChanged(de.uni_due	notifyClientChanged(de.u	1.0	0.5	0.875
notifyClientRemoved(de.uni_due	notifyClientRemoved(de.u	1.0	0.5	0.875
beginMeasurement(java.lang.St	beginMeasurement(java.l	1.0	1.0	1.0
addSubTask(java.lang.String)	addSubTask(java.lang.Stri	1.0	1.0	1.0
finishTask(de.uni_due.s3.sylage	finishTask(de.uni_due.s3.s	1.0	1.0	1.0
endMeasurement()	endMeasurement()	1.0	1.0	1.0
showError(java.lang.String)	showError(java.lang.String	1.0	1.0	1.0
showInternalError(java.lang.Stri	showInternalError(java.lar	1.0	1.0	1.0
createTaskStatus()	createTaskStatus()	1.0	1.0	1.0

Figure 4.9: In this screenshot of the experimental tool, the left column lists all installed OSGi bundles. For the component selected as requiring component in this column, in the upper third of the middle column the bundles are shown with their matching value as providing components. The lower rows of the middle column show the interfaces of the requiring components that are currently not bound at run time. The right column shows details how the matching value of the selected requiring and providing component was computed. The buttons on the lower right allow for an export of all or selected interfaces into a mapping properties file.

Graphical Assembly of Systems and Composites

Composite components consisting of several subcomponents may be composed with a tool that provides a component repository. An architect could select a set of components to be subcomponents of a composite and interconnect them within this composite, directly storing the composite in the repository for other architects to use. Such a tool could also be used to bind and unbind components at run time.

5 Evaluation

In this chapter the proposed component model is evaluated. The proposed model is applied to a case study, in which an existing application's architecture is adapted to a component architecture which is developed with the proposed component model. The experiences and problems during the case study are discussed in chapter 5.1.3. In chapter 5.2 the identified features of formal component models, the OSGi component model, and the proposed model are evaluated with criteria for the usefulness and applicability of engineering models defined by Selic [Sel03], and the results of this thesis are discussed.

5.1 Case Study

In the case study during this thesis, an existing application's architecture was adapted to a component-based architecture, implemented with the proposed component model. The adapted application is SyLaGen (Synthetischer Last-Generator) [BSGT03, SBG10], a performance measurement and evaluation tool developed at the working group “Specification of Software Systems” (S3) at the University of Duisburg-Essen.

5.1.1 SyLaGen

SyLaGen is an application for measuring the performance of client-server applications. The server is used to define a workload for a target system. The workflow consists of several weighted flows defining user behaviour. The clients are registered at the server for receiving commands for load generation, which is provided by the server in terms of the workload information and a class library to the clients. The class library is the so-called *adapter* and is used to access the target system. The workload defines how the client must use the adapter to generate the load. The clients consist of several worker threads for generating the load. Each worker thread simulates a user. A measurement in SyLaGen consists of a workload that is executed by the workers. When the measurement is finished, each worker sends the results of the measurement to the server.

SyLaGen uses three different measurement modes:

- Single: This mode starts a measurement with one walk through a flow per worker.
- Ecstasy: In this mode all workers generate as much load as possible by repeating to run flows for a predefined measurement time.

- Exploration: This mode can be used to test the scalability of a system. A measurement starts with a low number of workers. If the performance values of the target system do not reach a specified level, the number of workers is increased after a predefined time. When the specified performance values of the system are reached with a certain number of workers, the tool found the maximum number of simulated users for the desired performance.

Figure 5.1 shows the architecture of SyLaGen. The server is called *Master*.

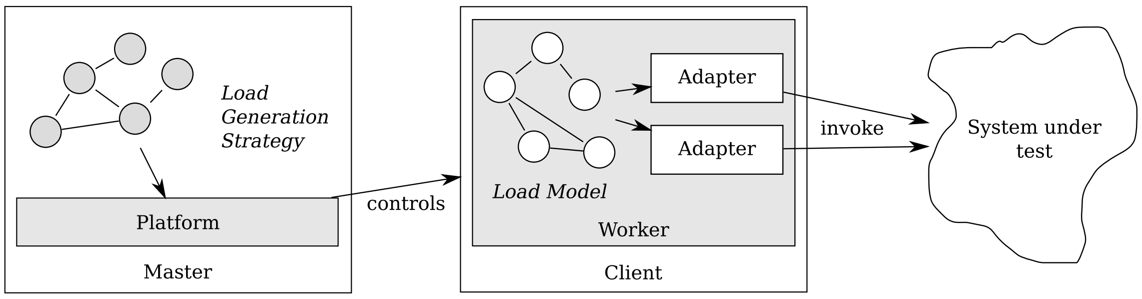


Figure 5.1: SyLaGen consists of a server, the so-called *Master* and clients. The Master offers a platform for a behavioural model representing a load generation strategy. Several clients are controlled by the Master. Each client has a number of worker threads which generate the load according to the load generation strategy submitted by the Master. The system under test can be accessed by using adapters which are Java libraries sent to the clients by the Master along with the load generation strategy. [SBG10]

The SyLaGen Master is implemented as an Eclipse [Ecl10b] application, making use of the features the OSGi Service Platform implementation Equinox [Ecl10c], which is the basis for the Eclipse Platform, and has also been used for implementing the proposed component model for OSGi. The SyLaGen Client is implemented as a Java Application. Both Applications share a common library, which contains the main data types and common parts of the communication functionality. The original architectures of the programs is depicted in figure 5.2. Both programs, the SyLaGen Master and client are implemented in a layered architecture. The Master consists of four modules. The module *common* defines most of the data types used in the system and includes functionality for network communication. The core module of the Master includes most of the Master's functionality. It uses the module *State Machine* for controlling the measurement. The user interface is used to control the Master. The client consists of three modules. The core module implements the main functionality, and relies on the module *common* for the most data types and network communication functionality. For generating load, an adapter implementation is given to the client.

The communication between Master and clients is managed by a proprietary communication library which strongly uses Java Architecture for XML Binding (JAXB) [Sun09a] for serializing and deserializing objects into XML documents.

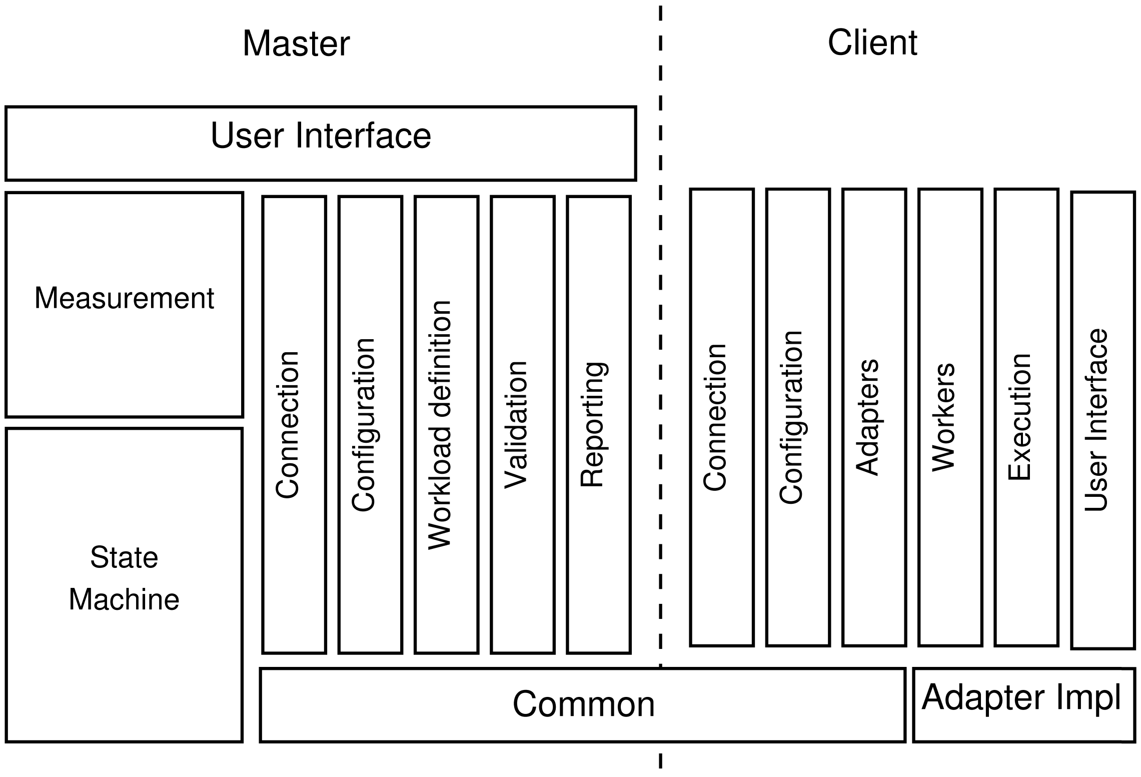


Figure 5.2: The SyLaGen Master consists of four modules. The core module implements most of the functionality. The client has three modules, of which the core module also implements most of the functionality. The module *common* is used in both applications.

5.1.2 Changes to the Architectures

Master and client are implemented as monolithic applications with a layered architecture. They share a library called *common*. Their structure had to be changed to represent a component architecture. The applications were split to an architecture where each component has a high cohesion with the goal to create loosely coupled components.

The Master's adapted architecture is divided into seven components as shown in figure 5.3. Since the master had a layered architecture, several services and types are used within many classes of the application. These services and types are now provided by their respecting components, and are defined as required services and types by the components using them. Several services and types were specified as common parameters for several components. These connections are indicated by the solid arrows in figure 5.3. The dashed arrows indicate connectors mapping with only required and provided interfaces. The component architecture of the client is shown in figure 5.4. The client application is divided into five components.

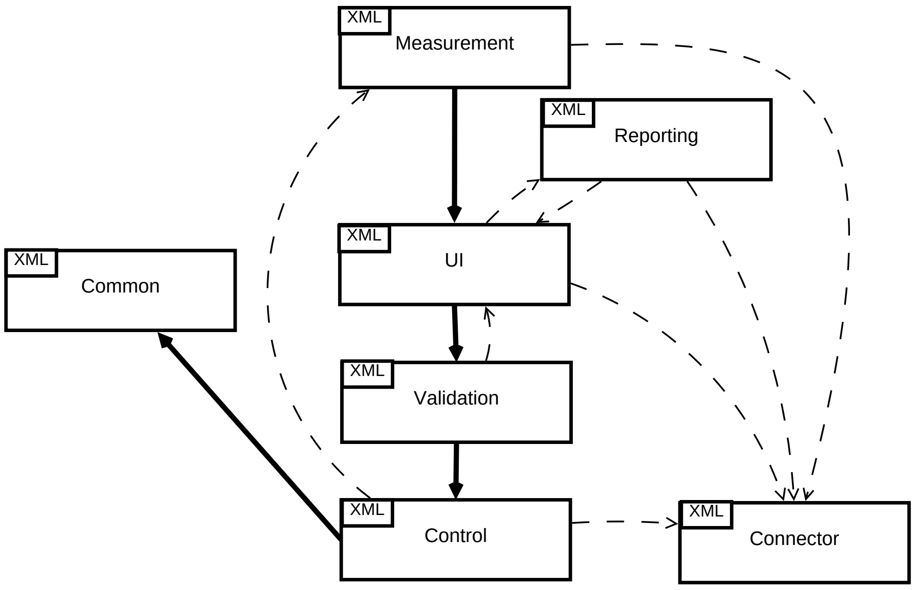


Figure 5.3: The adapted architecture of the Master consists of seven components. The connectors are represented by arrows. Solid arrows represent connectors handling mostly common parameters, dashed arrows indicate connectors handling just mostly required-provided relationships. The component *common* is reused in the client's architecture. The component interfaces are not notated in this figure.

5.1.3 Experiences and Problems

The implementation of the case study's application with the proposed component model indicated issues with the model's reference implementation, as some types were not mapped correctly. These issues could be resolved by considering a set of special cases, e.g. arrays as parameters of methods defined in provided or required

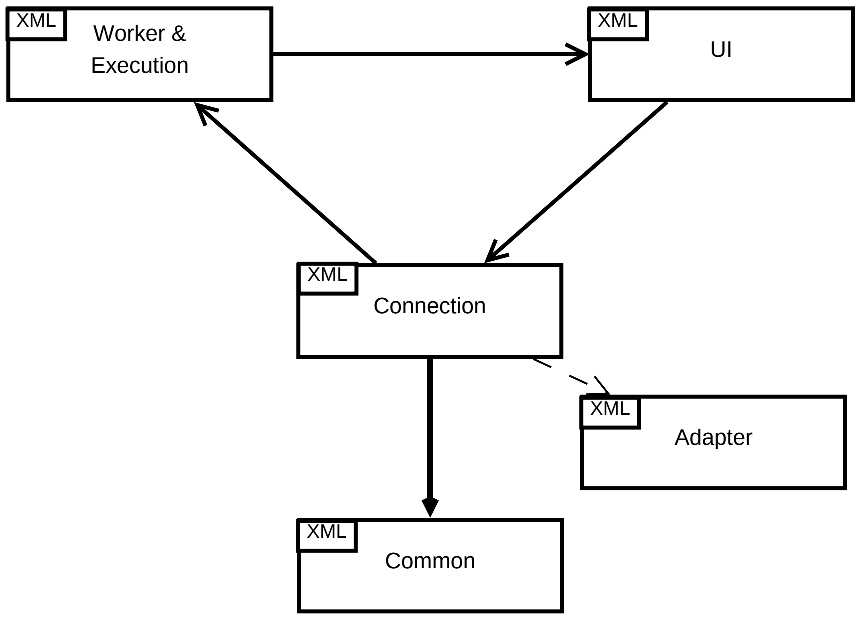


Figure 5.4: The adapted architecture of the client consists of five components. The component interfaces are not notated in this figure.

interfaces. These special cases were then considered in the class *DelegationInvocationHandler*.

All mapping files for the connectors between the components were generated with the experimental tool, which reliably suggested matching required and provided interfaces and their methods between components. The application of the experimental tool helped greatly to reduce the workload for creating method and interface mappings. E.g., the mapping file between the measurement and UI component consists of a total of 367 generated lines. Writing these mapping files manually would be a considerable workload. The estimated values for criteria weights, which are currently set as default values in the tool, reliably identified the interfaces and methods intended to match. However, the tool can still be enhanced. Some more special cases like generic types and array types as method parameters and return values cannot be matched perfectly. This drawback did not have an impact on the automatic interface and method mapping in the case study, but there is still room for improvement.

Adapting the architecture of SyLaGen to a component architecture proved to be harder than expected. As indicated in figures 5.3 and 5.4, an OSGi bundle had to be added to the Master's and the client's architecture, which holds the XML communication classes and all dependencies, including many data types used in the entire application. This bundle is represented as a rectangle in the upper left corner of the components in the figures. All components depend on this class library. The application heavily relies on JAXB for communication. The functionality of this connection could not be implemented in a component, because JAXB needs the implementation class definitions of the objects to serialize. If the data types had been defined as required or provided types, the objects given to JAXB would be instances of interfaces. These objects cannot be serialized by JAXB. Thus the XML library contains the communication functionality using JAXB and all necessary data types. As each component uses these data types, they must import the class library. Thus the components are not context-independent.

It would be possible to create a JAXB component that uses provided and the same required data types for serialization and deserialization, but its usage would be limited to exactly the provided data type objects. A universal JAXB component would need to rely on dynamically compiled classes from a universal required data type. While this would be possible, the JAXB component would have to extend the original JAXB functionality dramatically. This shows that existing frameworks and programming models might need to be considered in component models and their application. Patterns for different programming models and their representation in component models could help to systematically derive a component structure from existing applications and libraries.

Another architectural issue were cyclic dependencies. For example, when the user interface needs to be notified about changes in the data it shows, it has a dependency on the data types to be shown, and provides an observer service interface, which will be notified about changes in the data. Even when a control component is introduced, like it is known for the Model-View-Controller (MVC) pattern [GHJV94], the user interface component and the data have direct or indirect dependencies on each other. While there is no class-based dependency, there is still an architectural cycle, because the observer, known from the observer pattern [GHJV94], needs a reference to the observable, and the observable needs a reference to the observer.

A usual case of these dependencies led to method calls on required interfaces that take a provided interface as parameter. One example for that case is the user interface component, which requires a data type and provides an observer interface. The component containing the data type provides the data type interface and requires an observer to notify. In these situations method calls with a structure of *requiredDataType.create(providedObserverType)* are invoked, in which a provided type is given as parameter of a required method. These cases could be solved by declaring both interfaces, in this example the *requiredDataType* and the *providedObserverType* as common parameters, so both types are required and provided.

One initial goal for the implementation of the component model was Java 1.4 compatibility, for not constraining the compatibility of the OSGi platform. This goal could not be fulfilled. The mapping of Collection and Map classes with generics class parameters in the parameter or return type definitions of component interfaces raised the need to use APIs introduced with Java 5, for generics were introduced in this version. For example, a return type `HashMap<MyProvidedType>` needs to map the generic class parameter of the map. Without using the operations for generics, the implementation would not be able to see the interface type of the objects included in the instance of `HashMap`.

5.2 Discussion

In this section the results of this thesis are discussed. First, criteria for the usefulness and applicability of engineering models are applied to the features of formal component models identified in chapter 2, the OSGi component model described in chapter 3 and the proposed component model for OSGi defined in chapter 4. Then the advantages and disadvantages of the proposed component model compared to the OSGi component model are discussed.

5.2.1 Criteria for Model Evaluation

The proposed component model is an engineering model purposed for architectural descriptions of software systems. Selic describes five key characteristics for engineering models [Sel03]:

1. Abstraction: The model should provide an abstract view on the actual system, hiding unnecessary information. This abstraction should help focusing on the relevant parts.
2. Understandability: The model should provide the contained information in an intuitive way, reducing the intellectual effort needed to understand the facts represented by the model.
3. Accuracy: The information carried by the model should be true with respect to the modeled system.
4. Predictiveness: The model should provide a possibility to predict properties of the real system that are not obvious. Prediction can e.g. be achieved through formal analysis of the model or experimentation.
5. Inexpensiveness: The development of the model must be cheaper than the development of the actual system.

Abstraction

Abstraction is achieved by several concepts of formal component models. First, in formal component models the implementation of primitive components is typically hidden, providing an abstract view on the actual system. Additionally, composite components allow for specifying greater building blocks with hidden implementation details. These concepts provide an abstract view on the system.

An architecture in the OSGi component model consists of several bundles that import and export packages, and service objects that are published under an interface name in a service registry. OSGi does not provide concepts for abstraction. For understanding the architecture of an application, each component has to be inspected separately for interconnections. Composite components cannot be defined.

The proposed component model provides abstraction of the system architecture by explicitly specifying the component interfaces in a component configuration file. The actual implementation of components is not necessary to define an architecture. Composite components allow for hierarchical abstraction of the architecture, hiding the implementation details of subsystems.

Understandability

The main information carried by formal component models is the component specification and their interconnection. In the actual system, this information is not explicitly stated. Component interconnection may consist of many classes, interfaces and associations or class hierarchies. By hiding these details, the understandability is increased. However, as different component models have distinct representations of components and their interconnections, the understandability varies between the models.

The information provided by the OSGi component model are distributed in the source code and the bundle manifest file. While the import and export of packages is directly defined, the usage of these interconnections and the structure of the overall architecture are considerably hard to understand, as the information is widely distributed.

The component specification in the proposed model is given in a component configuration file, containing the necessary information for understanding the architectural role of the component in XML. The interconnections of subcomponents of a composite are defined in the configuration file of the composite, along with their mapping file. These mapping files are simple properties files. As they might become very large, they are not very intuitive, and finding a specific piece of information in these files can be hard.

Accuracy

The formal component models considered in this thesis are used to define an architecture prior to its implementation. While some implementations of these models allow for a code generation, changes in the code that influence the interconnection or specification of components are usually not reflected in the model. The examined formal component models do not provide means to check whether the actual system complies with the architecture.

The information provided by the OSGi component model is used by the runtime engine to interconnect the bundles and share objects using the service registry. However, this information does not describe the actual architecture at run time, as the source of imported packages is e.g. not explicitly defined. The source of a service object can also not be identified unambiguously.

The proposed component model accurately specifies components to their context, by explicitly naming required and provided services and types. However, the component interconnections in a system assembly is not necessarily described. While composite components define their interconnections within the component configuration file, the possibility to interconnect components using the Equinox command line allows for architectures that are not specified in a document or file, but only exist at run time.

Predictiveness

As formal component models have different foci, their predictiveness varies. Palladio can be used to predict performance properties for example. As KLAPER is an intermediate language for systematically generating analysis models from component models, it can be indirectly used for predicting several properties, including performance. Formal component models are used to describe component architectures prior to their implementation. Thus the functional correctness of component interconnections can be evaluated before they are actually bound. The formal founding of these component models also allow for formally checking interconnections. In addition to checking structural compliance, Π allows to formally check the compliance of two behavioural specifications in terms of path expressions.

The OSGi component model does not provide means to predict whether an architecture has specific properties or if it is applicable. As the OSGi component model does not provide information about component interconnection prior to run time, predictions are impossible.

In contrast to the OSGi component model, the proposed model allows to define an architecture without actually implementing it. Thus the functional applicability of an architecture can be predicted by checking the interface and method mappings of the connectors. The compiled Java interfaces provide formal information about the architecture together with the component and connector specifications. This information can be used for formally checking the satisfaction of the requirements and the applicability of interconnections, prior to a system's implementation. The behavioural description of component interfaces based on interface automata additionally allow to predict whether the components' usage assumptions for their interfaces are fulfilled by the interconnection.

Inexpensiveness

Except Π and UML, none of the considered component models allow for a detailed specification of the implementation. They do not need implementation details for modeling an architecture. Thus an architecture in a formal component model is less expensive than an architecture implemented in source code, because the formal description is an abstract view on the system, hiding information that is unnecessary when an architecture is assembled.

The architectural information in the OSGi component model is distributed in the implementation of the bundles and in their manifest file. For developing an architecture with OSGi, the system has to be implemented.

With the proposed component model, an architecture can be specified prior to the system's implementation. Instead of defining the software architecture by classes and their associations in the source code, each component and their interconnections can be specified in configuration files and by implementing their required and provided interfaces. Hence the specification of an architecture in the proposed model is less expensive than specifying the architecture in the implementation.

5.2.2 Evaluation Summary

The proposed component model has several advantages to the OSGi component model. For example, it allows for specifying the architecture prior to implementation, and provides a better abstraction than an architecture implemented with OSGi. As a result, the functional correctness of the architecture can be checked with tools, by finding unsatisfied requirements before the system is implemented.

However, the proposed component model does not provide the abstraction and understandability of formal component models. When a system in the proposed model is assembled by using the command line, the component interconnections are not traceable. Formal component models then provide a better understandable, abstract view on the complete system.

The considered formal component models in contrast lack the accuracy of the proposed model if they are not used by the runtime as actual architecture definition. When the models are used for description only, or for code generation, the compliance of the system with the architecture is not ensured.

While the proposed component model lacks some of the advantages of formal component models, it states the architecture more accurately and explicitly than the OSGi component model, and it provides more means for prediction of the architectures functionality. The proposed model also provides a better understandability by using a single point of configuration per component. The abstraction of implementation details for primitive and composite components is also not provided by the OSGi component model.

6 Related Work

This chapter provides information about work related to the content of this thesis. Formal component models are self-contained languages which may define a representation in the Java language. Several approaches to provide richer modular or component concepts in Java exist. In contrast to formal component models, these approaches aim to specify component semantics for the Java language. The approaches can be roughly categorized in two categories: language extensions and frameworks. In this chapter, some of these approaches are presented and their scope is defined.

6.1 Frameworks

The Java Enterprise Edition (JEE) [Sun09b] is the standardization of an enterprise platform for the Java programming language. JEE allows to write server-side applications. The widely-known component model of the JEE specification are Enterprise Java Beans (EJB) [Sun09c]. The component type in EJB is called a *Session Bean*. Dependencies in EJBs are declared by Java interfaces and annotations. An instance of the desired component is then injected by a framework directly to the field in the requiring class. The model also uses a shared interface approach, which leads to tightly coupled component through type definitions. Thus context independence is not possible. Additionally, this component model facilitates a flat component hierarchy, as no composite components exist. Behavioural constraints may be implemented with so-called interceptors that may intercept each call to a session bean. However, behavioural constraints cannot be specified descriptively for checking the compliance of two constraint definitions.

The OSGi Blueprint Container, which is specified in the OSGi Service Compendium [OSG09b], is a standardization of Spring Dynamic Modules [Spr10]. The Blueprint Container provides a dependency injection framework. It creates instances of service objects and injects them into a service user using a shared interface. This adds a third party, the injection framework, to the service model, but does not semantically enhance the OSGi service concept. The OSGi Declarative Services, which are specified in the same document as the OSGi Blueprint Container, and Apache Felix iPOJO [Apa10] are different approaches to this concept.

CORBA and its CORBA Component Model (CCM) [Obj06] is a practice-driven approach for representing components in programming languages. In the CCM, components are defined in an own language, which is independent from the application's programming language. With CORBA it is possible to create and interconnect components written in different programming languages. CORBA defines a language independent framework that allows for communication between its components. Components have required interfaces, but no connector exists as first class entity to interconnect components. Thus components are not context-independent. In addition, hierarchical component structures are not realizable with the CCM.

6.2 Language Extensions

ArchJava [ACN02] is an ADL expressed in an extension of the Java programming language. In ArchJava the architecture is defined with the application source code. Components in ArchJava are declared similar to classes, but use the *component* keyword instead of *class*. A component also defines required and provided *ports*, i.e. a collection of methods. *Connect* statements are used to interconnect the ports of two components. Components in ArchJava communicate through ports, which must be binary compatible. Thus context independence is not given in ArchJava. Also, as ArchJava is an extension to the Java language, the ArchJava compiler is needed to compile the programs developed with this technology. Thus a program cannot be part of an architecture without being recompiled by the ArchJava compiler.

Jiazzi [MFH01] is also a component system for Java. The concept of Jiazzi differs from the usual component concept in that the means of communication between component are not interfaces but instead whole packages including abstract Java classes are shared between the components. Each component defines so called *atoms* that contain references to *package signatures* describing the content of the referenced package. The imported classes can be instantiated by the importing component. The instantiation of an abstractly defined imported class results in an instantiation of the concrete class bound to this abstract requirement. This bound class is a subclass of the shared one. The abstract classes contained in those packages can also be used by the importing component by creating own classes that inherit from the imported ones, to create own instances of the shared class. As the components need to share packages, complete context independence is not possible.

7 Conclusion

Component-based software engineering (CBSE) and adjacent topics have been subject to research for decades. Creating software architectures from scratch using one of the several academic component models seems to be well understood. As some of these component models are even formally founded, these languages allow for simulation or analysis of the software architecture before the system is built. Though the component-based architecture concept is well-understood and backed up by formal reasoning over attributes, the benefits of the long research can hardly be used in practical component frameworks in modern programming languages. The current languages and frameworks do not leverage the research results of the past years.

This thesis aims at resolving this problem by first comparing formal component models to identify the core features. The identified features are compared to the component model of the OSGi Service Platform, a practice-driven module framework that is considered a standard component framework for Java. The results of this comparison are used to create a proposal for a new component model for the OSGi Service Platform that implements the identified features of formal component models. The proposal and its reference implementation is evaluated in a case study that shows the applicability as well as some weaknesses of the proposed component model.

The proposed component model has several advantages over the original OSGi component model. In the proposed model, the architecture is clearly stated and thus traceable, which improves understandability. Abstractions through composite components allow for efficiently specifying even very large systems, without losing the understandability of the complete architecture. In contrast to the original OSGi component model, the proposed model allows to specify an architecture prior to the system's implementation.

While this thesis focuses on the functional aspects of software architectures, the specification of quality attributes is not considered. This could be addressed in future work. The experimental tool which was developed during this thesis covers the generation of connector configuration files and finds unbound requirements of single components. Other starting points for tool support which were introduced in chapter 4.3 and include formal checking of interconnections and their behavioural descriptions, as well as tools for graphically composing component hierarchies, are also left for future work.

In conclusion, this thesis provides a deep examination of the main features of formal component models, and a starting point for further comparison with practice-driven component frameworks. The initial goal of adapting the component model of a framework to support the identified features could be fulfilled and confirmed in a non-trivial case study. In addition, this thesis deals with a technology that is currently strongly in the focus of Java development, as the integration of OSGi as a component standard in Java is currently under discussion by the Java community. Hence the results of this thesis might be considered in the development of component models for current or future programming languages.

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In *In Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367. Springer Verlag, 2002.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[Apa10] Apache Foundation. Apache Felix - Apache Felix iPOJO, February 2010. http://felix.apache.org/site/apache-felix-ipojo.html, accessed at 16 March 2010.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What characterizes a (software) component? *Software - Concepts & Tools*, 19(1):49–56, June 1998.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BP04] Tomas Bures and Frantisek Plasil. Communication Style Driven Connector Configurations. In *LNCS3026, ISBN 3-540-21975-7, ISSN 0302-9743*, pages 102–116. Springer-Verlag, 2004.

[BSGT03] Reinhard Bordewisch, Bärbel Schwärmer, Michael Goedicke, and Peter Tröpfner. Lastsimulation für Anwendungsumgebungen in vernetzten IT-Architekturen. *Mitteilungen der GI-Fachgruppe MMB*, 43, 2003.

[CFGGR91] Joachim Cramer, Werner Fey, Michael Goedicke, and Martin Große-Rhode. Towards a Formally Based Component Description Language. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, pages 358–378, London, UK, 1991. Springer-Verlag.

[Chi98] Shigeru Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[ČHPR09] Ondřej Černý, Petr Hošek, Michal Papež, and Václav Remeš. SOFA 2 Component System: User's Guide, 2009. http://sofa.ow2.org/docs/pdf/users_guide.pdf, accessed at 9 December 2009.

[CS01] Philip T. Cox and Baoming Song. A Formal Model for Component-Based Software. In *Proc. IEEE Symposia on Human Centric Computing Languages and Environments*, 2001.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120, 2001.

[Ecl10a] Eclipse Foundation. Eclipse Modeling - EMF - HomePage, 2010. http://www.eclipse.org/modeling/emf/, accessed at 7 March 2010.

[Ecl10b] Eclipse Foundation. Eclipse website, 2010. http://www.eclipse.org/, accessed at 14 March 2010.

[Ecl10c] Eclipse Foundation. Equinox, 2010. http://www.eclipse.org/equinox/, accassed at 14 March 2010.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. [u.a.], 1994.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java^{TM}Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.

[GMRS08] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. KLAPER : An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In *The Common Component Modeling Example*, volume 5153/2008, pages 327–356. Springer Berlin / Heidelberg, 2008.

[GMS05] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 25–36, New York, NY, USA, 2005. ACM.

[Hop00] Jon Hopkins. Component Primer. *Communications of the ACM*, 43(10):27–30, 2000.

[How96] T. Howes. A String Representation of LDAP Search Filters, RFC 1960, June 1996. http://www.ietf.org/rfc/rfc1960.txt, accessed at 23 March 2010.

[Inf03] Information Technology for European Advancement. ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices Project, Deliverable 1.5 - Revised specification of framework and models, July 2003. http://www.hitech-projects.com/euprojects/robocop/deliverables.htm, accessed at 2 February 2009.

[ISO06] ISO. Information technology - Programming languages - C#, Sep. 2006. ISO/IEC 23270:2006(E).

[Kar09] Karlsruhe Institute of Technology (KIT) and Research Center for Information Technology (Forschungszentrum Informatik, FZI). Palladio Component Model - Wissensbasis, December 2009. http://sdqweb.ipd.kit.edu/mediawiki/index.php?title=Palladio_Component_Model&oldid=17844, accessed at 23 March 2010.

[MBG10] Marco Müller, Moritz Balz, and Michael Goedicke. Representing Formal Component Models in OSGi. In Gregor Engels, Markus Luckey, and Wilhelm Schäfer, editors, *Software Engineering*, volume P-159 of *LNI*, pages 45–56. GI, February 2010.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. *Lecture Notes in Computer Science*, 989:137–153, 1995.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-Age Components for Old-Fasioned Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–222, New York, NY, USA, 2001. ACM.

[Obj06] Object Management Group. CORBA Component Model Specification, Version 4, April 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-01, accessed at 30 December 2009.

[Obj08] Object Management Group. Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1: CORBA Interfaces. Technical report, Object Management Group, January 2008. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF/.

[Obj09a] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2, February 2009. http://www.omg.org/spec/UML/2.2/Infrastructure, accessed at 22 December 2009.

[Obj09b] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2, February 2009. http://www.omg.org/spec/UML/2.2/Superstructure, accessed at 22 December 2009.

[OSG09a] OSGi Alliance. OSGi Service Platform Core Specification Release 4, Version 4.2, June 2009.

[OSG09b] OSGi Alliance. OSGi Service Platform Service Compendium Release 4. Version 4.2, Agust 2009.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. http://sunnyday.mit.edu/16.355/parnas-criteria.html, accessed at 26 March 2010.

[Pro10] ProSyst Software. OSGi services and embedded Java within home, vehicle and mobile devices, 2010. http://www.prosyst.com/index.php/de/html/content/132/Products-List/, accessed at 7 March 2010.

[RBH$^+$07] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, University of Karlsruhe (TH), May 2007.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[SBG10] Michael Striewe, Moritz Balz, and Michael Goedicke. SyLaGen - An Extendable Tool Environment for Generating Load. In Bruno Müller-Clostermann, Klaus Echtle, and Erwin Rathgeb, editors, *Proceedings of "Measurement, Modelling and Evaluation of Computing Systems" and "Dependability and Fault Tolerance" 2010, March 15 - 17, Essen, Germany*, volume 5987 of *LNCS*, pages 307–310. Springer, 2010.

[SDK$^+$95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.

[See87] S. Seehusen. *Determination of Concurrency Properties in Modular Systems with Path Expressions*. Phd thesis, University of Dortmund, Department of Computer Science, 1987. (in german).

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Softw.*, 20(5):19–25, 2003.

[SG94] Harald Schumann and Michael Goedicke. Component-Oriented Software Development with PI. Technical Report 1/94, Specification of Software Systems, Department of Mathematics and Computer Science, University of Essen, 1994.

[Spr10] Spring Source. Spring Dynamic Modules for OSGi Service Platforms, 2010. http://www.springsource.org/osgi, accessed at 16 March 2010.

[Sun07] Sun Microsystems. JSR 291, Dynamic Component Support for JavaTM SE. Technical Specification, 2007. http://jcp.org/en/jsr/detail?id=291, accessed at 30 December 2009.

[Sun09a] Sun Microsystems. JSR 222, Java Architecture for XML Binding (JAXB) 2.0. Technical Specification, 2009. http://jcp.org/en/jsr/detail?id=222, accessed at 23 March 2010.

[Sun09b] Sun Microsystems. JSR 316, Java Platform, Enterprise Edition 6 Specification. Technical Specification, 2009. http://jcp.org/en/jsr/detail?id=316, accessed at 23 March 2010.

[Sun09c] Sun Microsystems. JSR 318, Enterprise JavaBeans, Version 3.1. Technical Specification, 2009. http://jcp.org/en/jsr/detail?id=318, accessed at 23 March 2010.

[Sun10] Sun Microsystems. Dynamic Proxy Classes, Java SE Documentation, January 2010. http://java.sun.com/javase/6/docs/technotes/guides/reflection/proxy.html, accessed at 7 March 2010.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Essen, 26. März 2010

. . . .