

INFORMATIK BERICHTE

337 - 5/2007

Simplifying and Unifying Composition for Industrial Component Models

Ursula Scheben



FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

Simplifying and Unifying Composition for Industrial Component Models

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
Fakultät für Mathematik und Informatik der
FernUniversität in Hagen

vorgelegt von
Dipl.-Inform. Ursula Scheben
Hagen, Oktober 2006

Abstract

In a world of high productivity with a focus on maximizing profit, it becomes extremely important to produce new, reliable software as quickly and cheaply as possible. Component based software development (CBD) promised to reach this goal. By just composing prefabricated, reusable, well-tested components to new applications, application development should become much more productive and reliable. Application programmers should be able to select from a set of suitable components potentially produced by different vendors.

Industry contributed to this goal by introducing several different component models as e.g. JavaBeans, Enterprise JavaBeans (EJBs), the Component Object Model (COM), the Corba Component Model (CCM), and most recently .NET. Each component model defines its own standard for component look up, instantiation, access to the functionality of the component, communication, composition techniques available etc. Although these component models already are a big step towards the goals of CBD, especially the composition techniques supported are still too restricted, not simple enough, and vary from component model to component model.

Thus, the main goal of this thesis is to establish a basis and to develop techniques for simplifying and unifying the composition process for components belonging to industrial component models. We introduce a unifying component model comprising the main features of current industrial component models. This model provides some additional, useful features, as e.g. the support of bi-directional connections which can be established by a certain composition mechanism. Components of this model can be composed to yield components of a higher level of abstraction. These composite components are described by a composition language which supports late binding mechanisms through strict interface based programming. As components of industrial component models can be integrated into our unifying model, all its features are also available for existing industrial component models. Compositions can be checked for consistency based on a type system we define for our component model. This type system respects amongst others conditions for bi-directional connections and a certain kind of alias control. Besides consistency checking, the type system is used to decide whether a component can be replaced by another one without invalidating any existing composite referring to the component to be replaced. To further simplify the composition process, we focus on tool support for visual composition. Some useful features are introduced, as e.g. the guided establishment of needed interconnections and the colored depiction of constituents of a composite causing incorrect compositions.

Acknowledgements

I started my work at this thesis during the EU project EASYCOMP. I thank Prof. Dr. Poetzsch-Heffter for giving me the chance to participate in this project and for first discussions on the topic of simplifying the composition process.

Prof. Dr. Steimann supported me during the last two years in various ways. First of all, he became my Ph.D. supervisor although I started my work with Prof. Dr. Poetzsch-Heffter. Thanks a lot for doing so. In addition to all his duties he found time for intensive discussions on componentware and for proof-reading my thesis. He gave me valuable hints on how to improve it. Last but not least he relieved me from many tasks concerning teachings.

I am also indebted to Prof. Dr. Knoop for becoming my second referee. Because of his numerous tasks at the Vienna University of Technology and his research activities, his time is rare. Nevertheless he spent his free time to proof-read the 300 pages of my thesis and gave me valuable feedback. Furthermore he had always time for my problems and to encourage me. Many thanks.

I thank Monika Lücke for her ongoing support during the last years and the pleasant working atmosphere she disseminated. Thanks go to Ingolf Grebe for drawing several of my figures.

I would like to thank Prof. Dr. Jörg Roth and Dr. Daniela Keller for their warm response when I switched to the chair of programming systems and for reviewing my thesis. Especially Dr. Daniela Keller encouraged me several times to finish my work at this thesis. Although this year she got severe problems with her eyes, she carefully read several iterations of my thesis. In addition she relieved me of supporting several of our courses.

Special thanks go to my husband and my daughter for their ongoing understanding and encouragement. In the last months, my time for them was really rare. They forwent joint holiday and leisure. Without the support of my husband I would not have succeeded in finishing this thesis.

Contents

1	Introduction	1
1.1	A Vision of a Builder Tool	3
1.2	Contributions made by this Thesis	6
1.3	Organisation of the Thesis	9
2	Foundations	11
2.1	Terminology	11
2.2	Component Models	15
2.2.1	JavaBeans	17
2.2.1.1	Component Model	17
2.2.1.2	Composition Techniques and Consistency	22
2.2.1.3	Type System	25
2.2.1.4	Type Metadata	26
2.2.1.5	Compatibility / Substitutability	29
2.2.2	Component Object Model (COM)	30
2.2.2.1	Component Model	30
2.2.2.2	Composition Techniques	37
2.2.2.3	Type System	40
2.2.2.4	Type Metadata	44
2.2.2.5	Consistency / Correctness of a Composition	48
2.2.2.6	Compatibility / Substitutability	48
2.2.3	.NET	49
2.2.3.1	.NET Framework	49
2.2.3.2	Composition Techniques	60
2.2.3.3	Type System	64
2.2.3.4	Type Metadata	65
2.2.3.5	Consistency / Correctness of a Composition	67
2.2.3.6	Compatibility / Substitutability	67
3	Improvements over Existing Approaches	68
3.1	Component Models	68
3.2	Composition Techniques	73
3.2.1	Industrial Component Models	73

3.2.2	Visual Assembly	75
3.3	Type Systems for Components	77
4	Our Approach	82
4.1	The Unifying Component Model (UCM)	85
4.1.1	Basic Component Model	85
4.1.1.1	General Basic Concepts	85
4.1.1.2	Plug, A Higher Level Concept	99
4.1.1.3	Constraints on Connections	106
4.1.1.4	Summary of Concepts	109
4.1.2	Component Interface Specifications	110
4.1.2.1	General Specifications	110
4.1.2.2	Additional Specifications supporting Automatic Connections	117
4.2	Composition	118
4.2.1	Interconnections between Component Instances	119
4.2.1.1	Connections via services	119
4.2.1.2	Connections via plugs	120
4.2.2	Hierarchical Composition	120
4.2.3	More Details on Interconnections and Exports	126
4.2.3.1	Details on Interconnections	127
4.2.3.2	Details on Exports of Plugs	129
4.3	Used Type System for UCM-Components	132
4.3.1	Type Definitions	132
4.3.2	Subtyping	138
4.3.2.1	Subtyping of Services	138
4.3.2.2	Subtyping of Plugs	140
4.3.2.3	Subtyping of Component Interfaces	145
4.4	Correctness of a Composition	151
4.4.1	Interconnections between Component Instances	152
4.4.2	Export of Services and Plugs	157
4.5	Component Lookup	162
4.6	Instantiation of Composite UCM-Components	163
4.7	Helper Components	164
4.8	Compatibility / Substitutability	166
4.8.1	Polymorphic Component Instances	166
4.8.2	Replacing Components because of Upgrades or Change of Vendors	175
4.9	Realisation of Composite UCM-Components	176
4.10	Integration of Industrial Component Models	182
4.10.1	JavaBeans	184
4.10.1.1	Component Implementations and Component Interfaces Specifications	184

4.10.1.2	Integration of existing Concepts and Support for Services and Plugs	187
4.10.2	Component Object Model (COM)	190
4.10.2.1	Component Implementations and Component Interface Specifications	190
4.10.2.2	Integration of existing Concepts and Support for Services and Plugs	199
4.10.3	.NET	201
4.10.3.1	Component Implementations and Component Interfaces Specifications	201
4.10.3.2	Integration of existing Concepts and Support for Services and Plugs	207
4.10.4	Comparing Integration for the Various Component Models	211
4.11	Some Algorithms Supporting Visual Composition	213
4.11.1	Automatic Interconnections Using Plugs	213
4.11.1.1	Checking for Complementary Plugs	215
4.11.1.2	Service Mapping	219
4.11.2	Composing Plugs when Creating Composite UCM-Components .	221
5	Evaluation	223
5.1	The BPCE as a “Proof of Concept”	223
5.2	The CC-Builder	229
6	Related Work	234
6.1	Component-Oriented Languages	234
6.1.1	ArchJava	235
6.1.2	ComponentJ	237
6.1.3	Component Pascal	240
6.1.4	Others	244
6.2	Composition Languages	245
6.2.1	Bean Markup Language	246
6.2.2	Bean Plans	247
6.2.3	Piccola	249
6.3	Architecture Description Languages	251
6.3.1	Darwin	251
6.3.2	Acme	253
6.3.3	Wright	256
6.4	The Unified Modeling Language 2.0	259
6.4.1	Components	259
6.4.2	Internal Structure of a Component	262
6.4.3	Comparison of the UML- and our UCM-Approach	267

7	Summary and Perspectives	275
7.1	Summary	275
7.2	Perspectives	279
7.3	Conclusion	281
A	Summary of Used Graphical Elements	282
A.1	Graphical Elements	282
A.2	Typical Diagrams	285
	Bibliography	287
	Index	297

List of Figures

1.1	Hierarchically Composed Components	5
2.1	Contents of a Java Archive	18
2.2	Contents of the Manifest File contained in the Archive of Figure 2.1	19
2.3	Interface Negotiation by QueryInterface	34
2.4	Interface Pointer	35
2.5	Interface Pointer as a Pointer to a C++ Object	35
2.6	Connection between a Client and a Connectable Object	38
2.7	Aggregation in COM	38
2.8	Collaboration between Outer and Inner Object in COM	39
2.9	Base Interfaces in the COM Registry	42
2.10	Single-file and Multifile Assembly (Source: MSDN-Library)	53
3.1	Different Configurations of a Wordprocessor Application	72
4.1	Component Instances with their Clients and Implementing Objects	86
4.2	Customer Form to enter Customer Data	87
4.3	Required Services with Connection Points	91
4.4	Packaging Machines	100
4.5	Model View Controller	102
4.6	Cooperating Objects in Compound Documents	105
4.7	Two other Views of the Customer Form	106
4.8	Example for Required Different Service Providers	107
4.9	Component Instances with several Required Services of the same Type, which have to be connected to a common Service Provider	109
4.10	Component Model	109
4.11	Component Interface Specification	111
4.12	Overlapping Plugs	113
4.13	Connections via Plugs	120
4.14	Example of a Composite UCM-Component	122
4.15	Implementation for Atomic UCM-Components	122
4.16	Implementation for Composite UCM-Components	123
4.17	Composite UCM-Component with its Component Interface	124

4.18	Subtyping of Plugs (IConnectionPoint is shortened to ICP)	141
4.19	Invalidated Composite UCM-Component	146
4.20	Component Interface violating Subtyping Rules for Required Services . .	147
4.21	Subtyping of Constraints	148
4.22	Component Interface violating Subtyping Rules for Constraints	148
4.23	Valid and invalid Constraints concerning Subtyping	149
4.24	Complementary Plugs	156
4.25	Exports	157
4.26	Exported Constraints	161
4.27	Pool of Component Interfaces and Component Implementations	162
4.28	Nested Instances of Composite UCM-Components	163
4.29	Composite UCM-Component with integrated Delegator Component . . .	165
4.30	Composite UCM-Component with integrated Multiplexer Component . .	166
4.31	Strong Subtyping between CI and CI'	167
4.32	Weak Subtyping between CI and CI'	168
4.33	Component Info Object	177
4.34	Instance of a Composite UCM-Component	179
4.35	Windows Registry for COM Interfaces	192
4.36	Examples for (not) allowed Interconnections	214
4.37	Example for Plugs to be checked whether they are complementary	217
5.1	User Interface of the BPCE	224
5.2	List of Matching Provided Services	226
5.3	Enabling or Disabling additional Connection Support	229
5.4	Hierarchically Composed Components	230
5.5	Selection of a Style	232
5.6	Motif-Style	233
5.7	Windows-Style	233
6.1	Black- and White-Box Views of a Component (taken from [UML05] with slight modifications)	261
6.2	Another Representation of the Black- and White-Box View of a Compo- nent (taken from [UML05] with slight modifications)	261
6.3	Wiring through Dependencies on a Structure Diagram (taken from [UML05])	262
6.4	Data Logging Component	262
6.5	Data Logging Component using Parts	263
6.6	Detailed View of the Data Logging Component	264
6.7	Data Logging Component using Parts, Ports and Connectors	265
6.8	Internal Structure of a Component containing other Components as Parts (taken from [UML05])	266
6.9	Composite UCM-Component CP_DataLogging with its Component In- terface CI_DataLogging	273

6.10 CP_DataLogging represented in UML Notation as a Composite Structure Diagram	274
A.1 Representation of a Component Interface	285
A.2 Representation of a Composite UCM-Component	286
A.3 Set of Component Instances	286

List of Tables

3.1	Main Concepts of Industrial Component Models	71
3.2	Conditions on Sybtypes	80
5.1	List of the Required Services of the Selected Source Component (left figure) Selection of a Service Provider, here <i>Chapter</i> (right figure)	225
6.1	BML-Operations	246

Chapter 1

Introduction

The idea to build new software applications using prefabricated components to increase productivity can be traced back to the late 60ies [McI68]. In 1968, McIlroy presented his vision of software components and a market for those components in his article “Mass-Produced Software Components”. In his approach, software components were still at the level of routines/procedures like sine functions or input/output conversions and were intended to relief programmers from standard routine tasks. Such components should be arranged in families, e.g. varying by their degree of precision, robustness or by time and space performance. This very fine grained view of a component shifted to more and more coarse grained components in the form of modules/libraries which could already provide complex functionality or large sets of related functionality. This functionality could be reused in a black box manner in various contexts. With the invention of object oriented languages, modules disappeared more or less and classes “became the better modules” [Szy98]. Indeed, classes became the major means to structure software and led to the “class as component view”. The mechanisms for data abstraction and information hiding provided by these languages were targeted to develop independent, reusable units of code, but the opposite came true; a lot of large, monolithic programs arose. It became evident that classes are too fine grained to be used as independent, reusable components with a high level of complexity. Instead, a set of cooperating classes is needed to provide a complex functionality. But good concepts for building large units of code encompassing several classes were missing in object-oriented languages.

At the same time, the key concept for reuse in object-oriented languages, inheritance, led to several severe problems:

- When a developer wants to reuse a class by inheritance, he has to know the internals of this class to use it properly.
- If a base class is changed, this has an impact on all of its subclasses. Their behavior may be changed in an unwanted manner leading to unexpected errors. This problem is known as the *fragile base class problem* ([Szy98] pp. 102 and [MS97]).

These problems led to a new generation of software components which incorporated the advantages of former generation components. They allow one to reuse components as black boxes without needing knowledge of their internals, to structure software according to functional aspects instead of focussing on data, and they avoid inheritance in favor of composition techniques.

This new generation of components is represented by various industrial component models each of them defining its own standard including a definition of what constitutes a component, how components can be accessed by their clients, how components can interact and how components can be composed. The most prominent are JavaBeans [Bro97], Enterprise JavaBeans (EJB) [DP00, EJB03], the Component Object Model (COM) and its immediate successors [EE98, COM95, DCO98, DCO96], the Corba Component Model (CCM) [Sie00, COR02], and most recently .NET [Wes02, Löw05]. In contrast to former components as e.g. modules, these component models allow components to be instantiated. Thus, in a running application, several instances of the same component may exist.

These industrial component models have already made a big step towards the main goal of component based software development (CBD) [NT95, NL97, Szy98, Gri98, SC00a, SPJF02]: to create new applications rapidly by just composing prefabricated, reusable, well-tested components which can be selected from a large set of suitable components potentially produced by different vendors. Although the composition techniques available by the industrial component models are already simpler than those of former generations, they do not yet reach the goal of CBD to simply stick components together. They are still too complex or too restricted, error prone, and vary from component model to component model. Some composition techniques still need the skills of experienced software developers.

Thus, one of the main goals of this thesis is to further simplify the key task, the composition process, and to guarantee correctness of a composition. Composition can become much easier if tools support the composition process visually. The tools hide the composition techniques provided by the component models that still need programming experience from the user. Amongst others, tools can help us to select the right components and to check new compositions for correctness. Thus, in the following we present our vision of a useful builder tool which supports the whole composition process. Before describing our tool we motivate why we want to support industrial component models.

If components belonging to known component models can be composed with our tool, the tool will be accepted much easier since many software developers already use one of these models. In addition, we can profit from the advantages of these models. There are already a lot of useful components available from different vendors which can be reused by our tool and can further be composed within our tool. Thus, these existing components need not be re-implemented or wrapped which would be an unacceptable effort. In addition, using existing component models, we can build upon their existing infrastructure which is rather complex and are not forced to develop such an infrastructure by ourselves. Thus, our second goal is to develop a unifying component model

comprising the main features of current industrial component models which allows us to integrate them easily.

1.1 A Vision of a Builder Tool

There already exist several tools which allow a programmer to select components from a palette, place instances of them onto a composition window, configure them by manipulating their properties using property sheets, and more or less visually wire them together to build a new application. Wiring in this context generally refers to event based communication. Events occurring at one instance cause actions on another instance. (For more details, see Section 3.2.2). Unfortunately, such tools assume the used components to be completely self-contained in the sense that instances of them can be created and accessed independently without any further action as e.g. a wiring prior to use.

In contrast to these tools, we also want to support the composition of components which are not fully self-contained, but depend on functionality provided by other components to fulfil their task. Such dependencies are inherent to layered architectures where one layer may only access its direct neighbors. Components residing in a higher layer usually call functionality of components residing in the next lower level to fulfil their task. We call such dependencies on a functionality provided by other components *mandatory required functionality* since the component which needs access to the other one can not work properly without the required component. An examples for such a dependency is e.g. a bank application component which needs access to a database component. Mandatory dependencies are not restricted to layered architectures. They may also appear in other architectures. In a model view controller architecture for example, the view-component must have access to the model-component to access the data it has to show. A wordprocessor component needs access to an editor component to allow users to enter their data etc.

A component can declare functionality to be provided to it by other components for the following other reasons:

- The component is extensible in some sense. This means, the component is able to include an extra functionality provided by some other component, but it does not depend on it to be available. An example for such a component could e.g. be a wordprocessor component without spellchecking functionality, but with a possibility to incorporate another component providing the spellchecking functionality.
- The component is the source for a certain kind of notification specified in terms of operations. These operations must be implemented by those components which want to be notified. The source component notifies other components by calling these methods.

As in the last two cases the component works properly even, if no component providing the declared functionality is connected to it, we call these dependencies on functionality provided by other components *optional required functionality*.

Our tool should support a user of a component in detecting all of its dependencies and allow him to resolve such dependencies by means of visual composition. Resolving a dependency is done by giving the requiring component access to a component which implements the required functionality. This process is referred to as *connection*. To enable our tool to detect component dependencies and to resolve them, the components we can compose by our tool also have to declare explicitly the functionality they require in addition to the functionality they provide.

As will be motivated in the remainder of this thesis, the provided as well as the optional or mandatory required functionality of a component is expressed in terms of named *service interfaces*, so-called *services*. The service interfaces declare this functionality in terms of operations and can be compared to Java interfaces. They only declare the signature of the operations, they do not carry any implementation. These services build the contract between the component and its environment on a syntactic level. As the functionality of a component can be rather complex, its functionality can be divided into a set of service interfaces. Our tool will refer to services by various features.

As will be shown in Section 4.1.1.2, there are a lot of situations, where *groups of services* are useful which are semantically related and need a connection to one single partner component. Often, such groups relate provided as well as required services and model a bi-directional communication between two parties. A typical example are callbacks in which a provided interface comes with a required one. Thus, it would be useful to be able to define such groups as entities for interconnections, such that those groups can be connected via one single operation from a user's point of view. This simplifies composition further. We introduce such groups of services in more detail in Section 4.1.1.2 and refer to them as *plugs*.

Another step towards simplifying the development of new applications is the possibility to reduce complexity by being able to hierarchically compose components to new components of a higher level of abstraction which can then be used to build the application. Thus, in contrast to most other tools, our tool should be able to compose components to new ones hierarchically as described below.

Hierarchical Composition: Our typical composition process to build new components from existing ones looks as follows.

A programmer selects components from the list of available components and places instances of them onto the composition window. Then he selects each of the component instances C residing in the composition window which has at least one required service. He selects one of the mandatory required services R of C by clicking on the representation of the service, a little button labelled 'REQ' on the right hand side of the component. The service name appears as soon as the mouse is moved over the button. Then he selects a fitting provided service of another component instance in the compo-

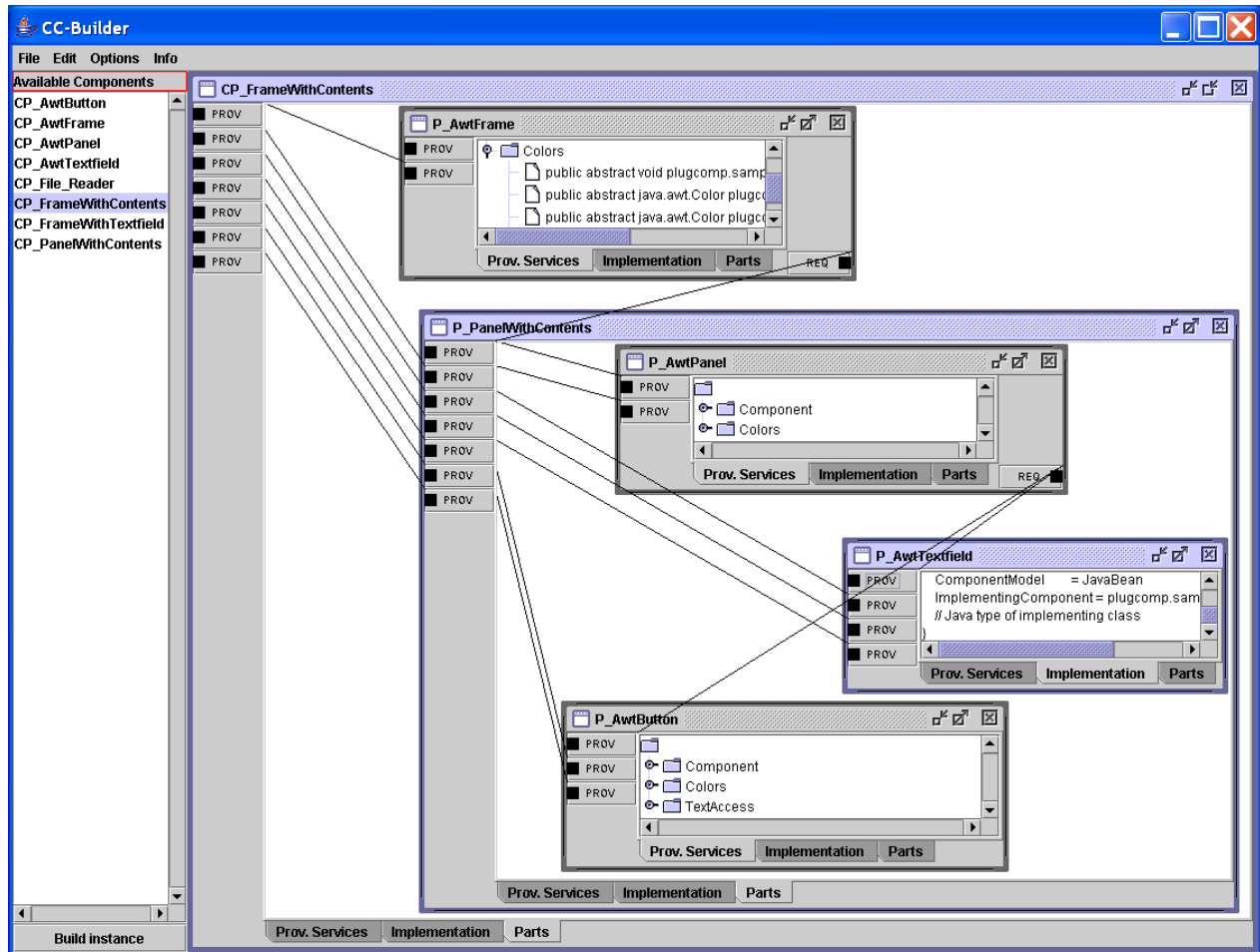


Figure 1.1: Hierarchically Composed Components

sition window by clicking on its button representation labelled 'PROV'. The connection is visualized by a line drawn between both services. The programmer continues this process, until all mandatory requirements are resolved except for those which are to be exposed by the new composite component just being built.

All optional required services the programmer wants to connect are connected using the same process. The programmer can save the new composition as a new component. He can select between the following two possibilities:

1. The tool creates a fitting component interface declaration for the new composite component automatically by a predefined algorithm.
2. The programmer selects each service he wants to be exposed explicitly and, if necessary or desired, declares a name for the service in the context of the new composite component. The tool then generates the corresponding service for the composite component and visualizes the export by a line drawn from this service to the service of the component instance exposing it.

The newly created component can further be composed to a higher level component. Figure 1.1 shows an instance of a composite component named “P_PanelWithContents” which consists of three parts and which is itself used to build a higher level component “CP_FrameWithContents”.

Besides hierarchical composition, an ideal tool has to provide a set of other features which are listed below.

Complete Tool Support: As our tool should support the whole composition process, it should at least

- provide the user with a tool box of components he can choose from and a composition window where the user can place instances of selected components for configuration,
- support the selection of components suitable to a requirement specified by the user or a component already selected for composition,
- allow a user to connect component instances residing in the composition window visually,
- support interconnections based on events components emit or consume, interconnections based on services the components provide or require, as well as interconnections based on plugs,
- support the creation of new components by means of hierarchical composition,
- support the creation of new applications,
- provide consistency checking for newly created components, and applications thereby distinguishing the handling of optional and mandatory required services,
- support the deployment¹ of newly created components,
- support the reconfiguration of existing compositions.

1.2 Contributions made by this Thesis

The main goal of this thesis is to establish a basis and to develop techniques for *simplifying and unifying the composition process especially for components belonging to industrial component models*. The thesis will focus on the following subgoals:

Goal 1: To treat different component models in a uniform way.

¹*Deployment* is the process of installing a component. After deployment, the component can be used for building new applications or composites.

Goal 2: To introduce mechanisms supporting especially bi-directional connections between component instances.

Goal 3: To develop techniques which allow the composition of components hierarchically thereby yielding new components of a higher level of abstraction.

Goal 4: To guarantee the correctness of a composition with respect to certain syntactical aspects.

Goal 5: To support the life cycle of components especially by simplifying the substitution of components.

Goal 6: To support especially visual composition.

These goals are reached by the contributions listed below. One contribution can help us to reach several goals. All goals contributed to are thus explicitly mentioned for each element in the list.

Contribution 1 (Development of a uniform component model) The essential concepts of existing industrial component models are subsumed in a uniform component model (UCM) in which the existing ones can be integrated. The unification focuses on the existing concepts essential for the creation and access of component instances as well as on their interconnections. This includes the concepts used to look up components, to instantiate them, to access their interface, to interconnect (wire) them etc. Thus, our focus is on the construction process and on accessing services. (Concepts like persistence, transaction or security are not regarded as they are not relevant for simplifying composition.) Besides unification, the developed component model offers several new concepts including e.g. plugs and constraints on interconnections controlling aliasing. Plugs are entities grouping semantically related services and especially simplifying bi-directional connections between component instances. Existing component models can be enriched with the new concepts in a simple manner.

Contribution 1 thus helps us to reach goals 1 and 2.

Contribution 2 (Development of a hierarchical composition technique) Several component instances may be aggregated and wired together to build a new, higher level component exporting dedicated service interfaces and plugs of its constituents. This technique can be used for the mentioned industrial component models even if these component models are flat ones. We introduce a simple language to describe our composite components. This language can be used by programmers to implement new composite components directly. It can also be used by tools to generate a composition description automatically from a set of component instances a user selected and wired together visually. In addition to other languages, our language supports interconnections and exports based on plugs and the integration of different industrial component models. The language strictly

separates component interface descriptions and descriptions of composite components. Component instances belonging to an instance of a composite component are strictly typed by their component interfaces only. Components to be used for instantiation need not be mentioned in the description of the composite component. Because of this fact, it is possible to leave the choice of a suitable component to be used for instantiation to the runtime system.

Contribution 2 helps us to reach goals 3 and 5.

Contribution 3 (Introduction of a type system for our uniform component model) A type system is needed which allows one to decide

- when and how component instances can be interconnected,
- whether a composition is correct,
- when a component can be substituted/replaced by another one.

For this purpose, a type system for components including plugs and constraints on interconnections is introduced. Besides the integration of plugs and special constraints, the type system differs from existing approaches in the handling of services a component needs from other components to fulfill its task. Our definition ensures that a component already referred to in composite components can be substituted by a compatible one without affecting these composite components.

Contribution 3 helps us to reach goals 2 - 6.

Contribution 4 (Support for component substitution) The substitution of components is simplified through late binding mechanisms and automatic compatibility checks between two components. In this context, a component C_1 is compatible to a component C_2 , if C_2 can be replaced by C_1 without invalidating existing composite components already referring to C_2 .

Contribution 4 helps us to reach goal 5.

Contribution 5 (Provision of special features supporting visual composition) The composition process is simplified by features especially supporting visual composition. For example, constituents of a composite causing incorrect compositions are highlighted by a red rectangle surrounding their representations in a composition window. Other examples are the automatic establishment of interconnections between fitting plugs of two component instances and a guided establishment of needed interconnections etc.

A graphical representation for components as black boxes with their services and plugs as well as a graphical representation for compositions are introduced. These representations can be used by tools to depict components, their interconnections and composite components composed from other components.

Contribution 5 helps us to reach goal 6.

1.3 Organisation of the Thesis

Chapter 2 introduces the basic terminology used throughout this thesis and describes existing industrial component models. The description focuses on features essential for the creation and access of component instances as well as their interconnections. Different composition techniques provided by the industrial component models are discussed. Typing and subtyping aspects for components are presented which are needed to decide whether components may be substituted by new versions or by other components. Exact type and subtype definitions are rarely available for industrial component models. In such cases we present our own definitions which is also one of the contributions of this thesis.

Chapter 3 describes the similarities as well as the differences between industrial component models and identifies several of their shortcomings with respect to the component model as such, their type system, and the composition techniques available. In addition to the composition techniques provided by the component models themselves we shortly discuss existing tools which support visual composition. For every field discussed we finally present the improvements we want to achieve over the existing approaches.

Chapter 4 provides our approach to achieve these improvements. Our unifying component model is introduced that additionally supports bi-directional connections between component instances and provides a simple way to compose component instances hierarchically. The used composition process supports late binding of component implementations to component interfaces. The interface of a component is precisely specified as well as its type. Subtype relations are defined between component interfaces and the entities a component interface consists of. This allows one to check compositions for correctness and to decide if a component can be substituted by another component without affecting existing compositions. To demonstrate that our component model enables the integration of industrial component models, integration is described for selected industrial component models. Some algorithms are presented which enable automatic interconnections between plugs and the composition of plugs to greater ones. These algorithms are especially useful for visual composition and complete the tool support already described in Section 1.1.

As a proof of concept, Chapter 5 presents two tools, the *Bean Plug Composition Environment* and the *Composite Component-Builder*. Our first tool allows one to compose JavaBeans based on services especially supporting bi-directional connections. It demonstrates how JavaBeans can be used as atomic components in our component model and how the support for visual composition can look like. The second tool demonstrates how composite components of an arbitrary level of complexity can be built hierarchically and their provided methods called without knowing their provided services and service interface types in advance.

Chapter 6 relates our approach to other approaches in the area of component-based development: component-oriented programming languages, composition languages, architecture description languages and the UML2.0 approach concerning components.

The thesis concludes with a summary of the work at hand and outlines remaining future work.

Chapter 2

Foundations

This chapter introduces the basic terminology used throughout this thesis. Furthermore it describes the state of the art concerning industrial component models.

For the reasons mentioned in Section 1.1, we want to have a unifying component model and visual composition especially for industrial component models. We therefore have to study existing industrial component models rather detailed to know what can be unified and later on to verify that they can be represented by our new model. For every component model the definition of components, their lookup, their instantiation, the structure of a component interface in terms of its accessible entities, the access to these entities etc. are described. Possible component type and subtype definitions are considered even if exact type and subtype definitions do not exist in the literature. In this case we present our own definitions. Subsequently the kind of meta data available to describe and access a component are discussed. The main composition techniques are introduced as well as consistency conditions for compositions. The description for each component model ends with considerations concerning substitutability of components.

As our main focus is on providing basic techniques which allow one to unify and simplify the composition process and to check compositions for correctness and not primarily on certain visual support, visual composition is not considered in this section. Instead visual composition is described in Section 3.2.2.

2.1 Terminology

In the literature dealing with component based development (CBD) [NT95, NL97, Szy98, Gri98, SC00a, SPJF02], a lot of different definitions can be found concerning components, compositions and other terms. Szypersky [Szy98] for example defines components and composition as follows:

*“A **component** is a unit of composition with contractually specified interfaces and explicit context dependencies only.”*

*“**Composition:** Assembly of parts (components) into a whole (a composite) without modifying the parts.”*

Nierstrasz and Dami [NT95] state:

*“In short, we say that a **component** is a static abstraction with plugs. By static, we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By abstraction, we mean that a component puts a more or less opaque boundary around the software it encapsulates. With plugs means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages, etc.).... **Software composition**, then, is the process of constructing applications by interconnecting software components through their plugs. The nature of the plugs, the binding mechanisms and the compatibility rules for connecting components can vary quite a bit, as we shall see, but the essential concepts of components, plugs, plug-compatibility and composition remain the same.”*

In the following we introduce our own definitions of relevant terms used throughout this thesis.

Component: A *component* is some reusable piece of code implementing a certain functionality. A component can only be used as a black box. Access to the functionality of a component is only possible through a well-defined interface. Similar to a class, a component can be instantiated. We do not claim that components have to be built using classes. Their code may be based on procedural languages or on composition languages as well.

Component Instance: A *component instance* is a runtime entity conforming to the code of the component it is an instance of. In contrast to an instance of a class, a component instance normally consists of a **set** of collaborating objects together providing the functionality of the component. Some of these objects build the interface to the environment of the component and can therefore be accessed from the outside. Other objects belonging to the component instance can only be used internally.

Component Interface: The *component interface* defines the contract between the component and its environment. The functionality of the component can only be accessed via its component interface. A component interface can consist of a set of properties which can be changed from the outside, a set of operations which can be called from the outside, a set of events the component can emit, etc. Additionally, the set of operations the component needs from its environment to fulfill its task also belongs to the component interface.

Service Interface: The overall functionality of a component can be divided into a set of smaller functional units which can be accessed individually. The operations implementing such parts of the functionality of a component can be grouped by *service interfaces*. Thus the interface of a component can contain several service interfaces together providing the overall functionality. Similarly, the functionality a component needs from its environment to work properly can be divided into several service interfaces. A service interface is often simply referred to as *interface*.

Enabling Required Service Interface: As described in Section 1.1 a component may declare a service interface to be provided to it by other components to express that it is able to notify other components of certain events by calling the operations of this interface or that it is extensible by this interface. Extensibility means, that the component is able to include the extra functionality specified by the declared service interface that has to be provided to it by some other component. The component itself however does not depend on this functionality to be available to it. Similar to [SM05] we sometimes refer to these two kinds of service interfaces a component declares to be provided to it by other components as *enabling required service interfaces*. This is due to the fact that in the case of extensibility the clients of the component profit from the extensibility of the component and in the case of notifications, the components implementing this functionality profit from the implementation instead of the component declaring the functionality as required one.

Component Model: A *component model* essentially consists of rules specifying what kinds of components are supported, what constitutes a component, how the interface of a component looks like, how components can be accessed, whether components can be distributed, what composition techniques are primarily available, etc.

Connection / Wiring: Two component instances are *connected*, if they have a means to communicate somehow. Two examples of possible connections are event-connections and interface-connections. Two component instances are connected via a direct event-connection, if one component instance (target) receives events emitted by the other component instance (source). Indirect event connections use a mediator. The mediator receives the events instead of the target and in turn calls corresponding operations on the target. Two component instances are connected via an interface-connection, if one of the component instances has access to a service interface of the other component instance that is, it can call the operations belonging to this service interface. In the context of this thesis the term *wired* is used synonymously to *connected*.

Assembly: An *assembly* is a set of interconnected component instances¹.

¹In this thesis we focus on assemblies on instance level. The term assembly is sometimes also used on code level as e.g. for Enterprise JavaBeans.

Assembly Tool: An *assembly tool* is a tool to wire component instances visually.

Composite: A *composite* is a component instance, built from several other component instances.

Composite component: A *composite component* is a component, which is built from several other components.

Constituent: A component instance used to build a composite or a placeholder/variable for a component instance used to build a composite component is called a *constituent* of the composite / composite component.

Composition: The term *Composition* has two different meanings. First, composition denotes the process of constructing applications, components or assemblies from already existing components. Second, a composition is the result of a composition process that is, a composed application, a composite component or an assembly.

Composition Language: A *composition language* is a language which is targeted to describe a set of interconnected component instances by simple means. Such languages generally have first-class syntax and semantics to support composition operations.

Assembly Time: The time, where component instances are assembled/composed to build new applications, components or simply a web of interconnected component instances, is called *assembly time*.

Configuration Time: Each component should be capable to run in a range of different environments. Therefore, a component has to be adaptable to the needs of different customers. The time, where customization takes place, is called configuration time. This time is also sometimes referred to as *design time*. Often, configuration and assembly take place at the same time.

Deployment: *Deployment* is the process of installing a component. After deployment, the component can be used for building new applications or composite components.

Client: A *client* is a piece of software which calls operations provided by other pieces of software. A component calling operations provided by other components is therefore a client.

Server: A *server* is a piece of software which provides functionality to other pieces of software. A component providing functionality to others is therefore regarded as a server.

Static Dependency: *Static dependencies* on other components are hard wired in the code of a component and are assumed to be resolved at component instantiation time without problems. That is, components referred to in the code of another component are assumed to be deployed as needed so that instances of them can be created at runtime. Static dependencies allow component instances to create other component instances they need at runtime by themselves.

Dynamic Dependency: A component can specify that it depends on a (service) interface to be provided by another component not specified further in the program code of the requiring component. At component instantiation time, this reference is not yet resolved. A third party has to create a suitable component instance and to bind a fitting provided interface of this component instance to the open reference. This kind of dependency on another component is not resolved until runtime/assembly time and needs a third party to define this interconnection. This kind of dependency is called *dynamic dependency*.

Implicit Interface of a Class: With an *implicit interface* of a class we mean all public members declared by the class, as e.g. properties, emitted events and methods. These declarations are made in the context of the class. Generally, there is no extra interface declaring these members apart from the class declaration.

2.2 Component Models

Industry contributed to componentware by providing different component models from different companies or groups like e.g. Sun's JavaBeans [Bro97] and Enterprise JavaBeans (EJB) [DP00, EJB03], Microsoft's Component Object Model (COM) [EE98, COM95, DCO98, DCO96], OMG's Corba Component Model (CCM) [Sie00, COR02], Microsoft's .NET Framework [Wes02, Löw05] and WebServices [Wes02]. All these models can be used in the context of existing OO-languages. Instead of describing all available models we focus on JavaBeans, COM and .NET for the following reasons.

The JavaBeans component model is described, as it is a light-weight component model especially suited for visual composition. Components of this model are used as atomic components in our tools described in Chapter 5. Microsoft's COM is described, as it is one of the first, well-known component models independent from a particular programming language, allowing access to its services only through well-defined interfaces and avoiding inheritance. Although .NET is a framework rather than a component model, Microsoft is promoting .NET instead of COM. Therefore, and because .NET is one of the most recent approaches, we also have a look at .NET and focus on how components may be implemented using .NET.

The description is divided into six parts:

1. The first part titled *Component Model* essentially deals with the definition of components, their lookup, their instantiation, the structure of a component interface in terms of its accessible entities, the access to these entities etc. The contents of this part is structured in the same way for all component models described. These topics are essential for deciding whether and how components of these models may be integrated into our component model discussed in Section 4.
2. As components are for composition, composition techniques are of great importance. Thus, in part two, *Composition Techniques*, the composition techniques available are described. Composition is the process of constructing applications or components from already existing components. In traditional OO-programming, instances of components are created and accessed using special APIs or normal OO-constructs. This includes the creation of new components from existing ones where a part of the functionality of the used components is provided by the new one (*the composite*) using delegation mechanisms. Delegation means that the composite component re-implements operations already implemented by one of its used components. For its own implementation, the composite uses the already existing implementation by calling the corresponding operations on an instance of the used component.

One of the main goals of CBD was to simplify the creation of new software by using predefined components which had only to be stuck together. Therefore we focus on mechanisms allowing to more or less simply stick (connect) component instances together. This comprises composition techniques like event based or interface based connections or aggregation, but no delegation. For the component model under consideration the primary possibilities to connect/compose two component instances and the possibilities available to define sets of interconnected component instances are described. It will be discussed, how far these sets can be used as new applications or components.

It will be seen that most of the current industrial component models only provide a flat component model. That is, there are predefined means to connect two component instances, e.g. by events or service interfaces, but it lacks a possibility to define new components from existing ones by other techniques than using normal programming languages.

3. The third part titled *Type System* gives type and subtype definitions for the industrial component model under consideration. If precise definitions can not be found in the literature or definitions vary from author to author, we present our own definitions which is also one of the contributions of this thesis.
4. The fourth part, *Type Metadata*, is essential for tools (e.g. Visual Basic or Bean-Builder) which have to treat arbitrary *unknown* components. To handle arbitrary

components, a tool must have a means to learn about e.g. the operations a component supports, the events a component can emit or the operations a component needs from other components to fulfill its task. Additionally, a tool must be able to create component instances and to invoke their provided operations in a standardized way. For this purpose, a standardized kind of type meta information must be available for each component as well as a standardized way to invoke arbitrary operations which are both described for the component model under consideration.

5. Before being able to connect two component instances, one has to decide, whether these component instances really fit together. In this part, *Consistency / Correctness of a Composition*, it will be discussed, whether and how this can be achieved for the main composition techniques available for the component model under consideration. That is, the conditions for valid interconnections concerning the main composition techniques are discussed. Consistency checks only focus on interconnections between component instances, not on clients getting access to an entity of a component interface nor on whether all components needed for an application or composite component are available for instantiation. It is simply assumed that all components referred to by an application or composite component are available.
6. The last part titled *Compatibility / Substitutability* deals with the conditions which ensure that compositions referring to a component *C* can be retained without changes to their code if *C* is substituted by another component.

Parts two and five are combined where appropriate.

2.2.1 JavaBeans

2.2.1.1 Component Model

The JavaBeans component model was developed by Sun Microsystems in 1996. JavaBeans were especially designed to provide a light-weight component model used to implement small to medium sized controls. Instances of JavaBeans are intended to be visually customized by a builder tool at assembly time (in the JavaBeans specification referred to as design time) and to be wired together based on events. (For more details on wiring based on events please refer to Section 2.2.1.2.) Customization is done by setting publicly available properties as e.g. background color and fonts for visible components or an initial URL for non visible components. Beans are especially suited as GUI components but they may as well be used for data access etc. JavaBeans cannot be distributed, i.e. instances of JavaBeans may only communicate within the same virtual Java machine. They are platform independent because they consist of a set of Java class files (and resources) and Java is platform independent.

1. *Component:*

A JavaBean consists of a set of Java class files and a set of resources where needed. Resources are e.g. icons or serialized data used to instantiate a bean with predefined values. One of the Java classes constitutes the interface of the bean to its environment and is used for instantiation. This class is often referred to as the JavaBean itself, although the JavaBean comprises several other (helper) classes (see Figure 2.1). In the following, we will refer to this class as the *JavaBean-class*. JavaBeans are divided into visible and invisible beans. For visible JavaBeans, the JavaBean-class has to extend `java.awt.Component` so that instances of such beans can be added to visual containers.

An instance of a JavaBean is also often referred to as a bean. Therefore it depends on the context whether the component is meant or an instance thereof.

2. *Deployment unit holding a component:*

The classes and resources a JavaBean consists of are typically packaged into a Java Archive (JAR file), which is delivered as the deployment unit. Several beans may be packed into the same JAR file. A *manifest file* is added to provide information on the JavaBeans contained in the JAR file. The manifest file names the beans contained in the JAR file by marking a listed class or ser-file name representing a bean with `Java-Bean:True` as shown in Figure 2.2. (A ser-file represents a serialized bean instance.)

Figures 2.1 and 2.2 show the contents of the JAR file and the contained manifest file for the SorterBean shipped with Sun's Bean Development Kit (BDK).

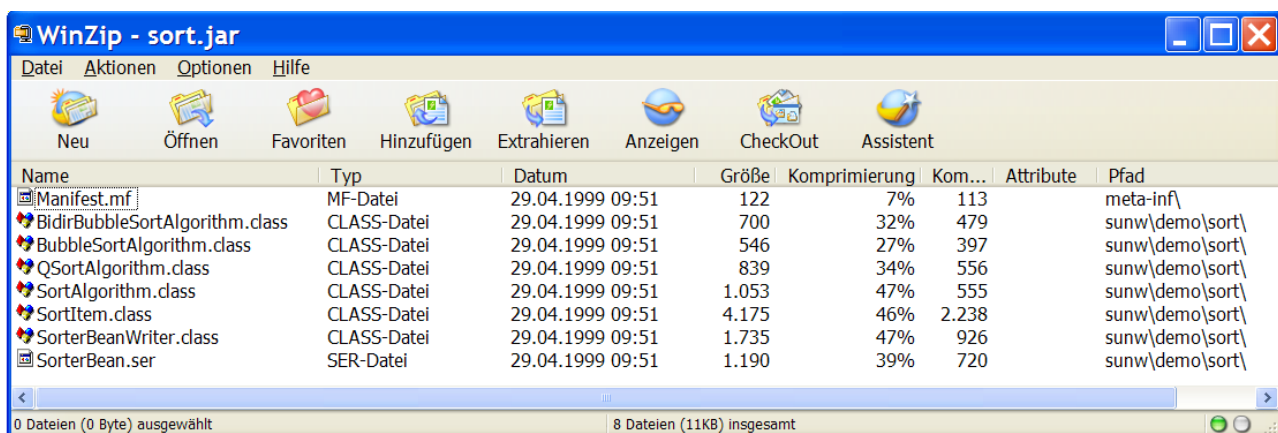


Figure 2.1: Contents of a Java Archive

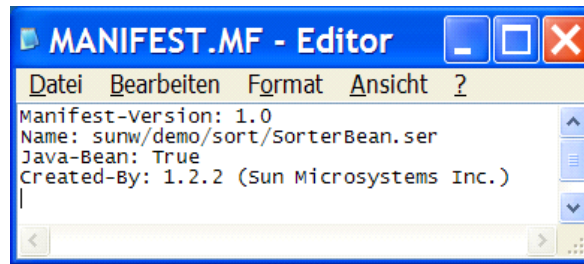


Figure 2.2: Contents of the Manifest File contained in the Archive of Figure 2.1

3. *Structure and specification of component interfaces:*

The interface of a JavaBean component to its environment is structured into a set of properties for customization purposes, a set of events which can be emitted or consumed by the JavaBean and a set of methods which implement the behavior of the bean. All properties, events and methods available for communication are declared in the JavaBean-class and its superclasses.

Properties: For the purpose of encapsulation, the properties are not defined by publicly available attributes, but by methods defining read- and/or write-access, so-called setter-/ getter-methods. To be able for a tool to determine the properties of a bean, these methods have to obey special naming and signature conventions:

```
public void set <PropertyName>(<PropertyType> value);
public <PropertyType> get<PropertyName> ();
public boolean is<PropertyName> ();
```

As an example, we show the signatures of the setter- and getter-methods of a property with name `background` of type `java.awt.Color` and the getter-method of a property with name `visible` of type `boolean`:

```
public void setBackground(java.awt.Color color);
public java.awt.Color getBackground ();
public boolean isVisible();
```

Events: JavaBeans can declare that their instances can be sources of specific types of events or that they can listen to specific types of events.

A source propagates event notifications to registered listeners by calling corresponding event notification methods on them (e.g. `mouseClicked`, `mouseEntered`). Each distinct kind of event notification corresponds to a distinct method. All methods belonging to a specific type of event are grouped by an *EventListener* interface (e.g. `MouseListener`). All *EventListener* interfaces have to inherit from `java.util.EventListener`. Potential listeners to a special type of event have to implement the corresponding *EventListener* interface.

Generally, event notification methods conform to the following pattern:

```
public void <eventOccurrenceMethodName>(<EventName>Event event);
```

where *<EventName>Event* denotes the type of the event state object encapsulating the state associated with the event notification (e.g. `MouseEvent`). This event state object is the sole parameter and its type inherits from `java.util.EventObject`. An example is `public void mouseClicked(MouseEvent evt);`

To know the listeners who want to be notified of the occurrence of a special type of event, a source has to provide corresponding registration methods. (De)Registration methods have to conform to the following naming and signature patterns:

```
public void add<ListenerType> (<ListenerType> listener);
public void remove<ListenerType> (<ListenerType> listener);
```

where *ListenerType* is the name of the `EventListener` interface grouping the event notification methods. As an example look at the registration methods for `MouseListener`:

```
public void addMouseListener(MouseListener l);
public void removeMouseListener(MouseListener l);
```

JavaBeans identify themselves as being sources of particular events by defining the corresponding registration methods obeying the naming and signature patterns from above. By implementing `EventListener` interfaces they identify themselves as being potential listeners to the corresponding types of events.

Therefore tools are able to identify possible event sources and the type of events they support based on the (de)registration methods. Potential event listeners are identified by the implemented listener interfaces.

The following example shows an `EventListener` interface with its event notification methods, the type of the event state object used as a parameter for the event notification methods, a class which is a source of the type of event defined by the listener interface, and a class implementing the listener interface thereby becoming a potential listener.

Example 2.2.1 (Event Sources, Listeners, and Event State Objects)

```

/*****      Type of the event state object      *****/
public class ControlEvent extends java.util.EventObject {
    // ...
}

/*****      EventListener interface      *****/
interface SimpleControlListener extends java.util.EventListener {
    void ControlFired (ControlEvent ce);
}

/*****      Source of Events      *****/
public class ControlContainer {
    private Vector controlListeners;
```

```

/* ----- Registering ControlListeners ----- */
public synchronized void addSimpleControlListener (SimpleControlListener cl){
    if (!controlListeners.contains(cl))controlListeners.addElement(cl);
}
/* ----- Firing of events ----- */
protected synchronized void fireControlFired(ControlEvent ce) {
    for (int i = 0; i < controlListeners.size(); i++) {
        ((SimpleControlListener) controlListeners.elementAt(i)).ControlFired(ce);
    }
}
}

/***** Potential listener *****/
public class SimpleController implements SimpleControlListener {
    // ...
    public void ControlFired (ControlEvent ce) { ... }
}

```

Publicly available methods: All other methods declared as public may have arbitrary signatures. They are used to invoke behavior implemented by the JavaBean.

To determine all properties, events, and publicly available methods of a bean, tools also inspect the superclasses of the corresponding JavaBean-class. For this purpose, tools can use a so-called *introspector*

(`java.beans.Introspector.getBeanInfo(Class beanClass)`, see Section 2.2.1.4, page 27). The introspector can only do its work, if the bean conforms to the above mentioned naming and signature rules.

If a bean does not conform to these rules or if it wants to specify additional features, like e.g. an icon representing it in a builder tool or a specialized *property editor* to be used to change one of its properties, the bean has to provide a *BeanInfo* class. The introspector can incorporate the information already provided by this BeanInfo class. The BeanInfo class has to implement the interface `java.beans.BeanInfo`. The BeanInfo class should be distinct from the JavaBean-class and its name should be the name of the corresponding JavaBean-class followed by 'Bean Info', e.g. `SorterBeanBeanInfo`. For more information on BeanInfo classes see Section 2.2.1.4 paragraph BeanInfo on page 27.

4. **Component lookup, instantiation and access:** The manifest file of a JAR file can be searched for all occurrences of Java classes which are marked by `Java-Bean: True`. These classes are the JavaBean-classes for the JavaBeans deployed by this JAR file. Knowing the name of the JavaBean-class, an instance of the corresponding JavaBean can be created by a call to
`java.beans.Beans.instantiate(ClassLoader cl, String beanName).`

As the actual parameter for `beanName` one has to use the fully qualified name of the `JavaBean`-class as string. In Java, *class loaders* are used to load classes into memory. For more details on this subject, please refer e.g. to

<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html>.

After the instantiation of a `JavaBean`, the client has access to an instance of the `JavaBean`-class defining the interface of the bean. Therefore the client has access to all properties of the bean instance by its setter-/getter-methods, to the listener registration methods and all other publicly available methods.

The access to properties and event registration methods is especially used by tools to configure/customize bean instances with property editors and to connect suitable event sources and listeners.

5. *Entities of the component interface as parameter or return value of a method:*

The only way to access the interface of a `JavaBean` is to get a reference to an instance of the `JavaBean`-class. There are no smaller entities to be referenced. As an instance of a `JavaBean`-class is nothing more than a pure Java object, references to bean interfaces can be passed as parameters to methods implemented by other beans.

2.2.1.2 Composition Techniques and Consistency

Composition by delegation and the creation and access of component instances by other component instances or applications is possible for the `JavaBeans` component model as for every other industrial component model. But the primary composition technique available for `JavaBeans` is the wiring of bean instances based on events. A set of bean instances is *assembled* to build an application or applet. The used bean instances interact by sending and receiving events. The receiving bean instances react according to the event notifications obtained. To enable this kind of interconnection/wiring, event sources and event listeners are defined as already described in Section 2.2.1.1 item 3. A bean instance interested in listening to a special type of event has to implement all event notification methods corresponding to this type of event. Such an instance is then registered at a 'suitable' event source as a listener. If an event occurs, the event source notifies all listeners registered for this type of event by calling the corresponding event notification methods on them.

In this context 'suitable' means that source and listener have to fit together as summarized in the following consistency condition:

Condition 2.2.2 (Valid Event Connections) *Let L be an event listener implementing an `EventListener` interface `EListener` and let S be an event source which fires events defined by the `EventListener` interface `ESource`. L can be registered at S , if `ESource` is equal to or a supertype of `EListener`.*


```

public synchronized void removeSimpleControlListener
    (SimpleControlListener cl) { ... }

/* ----- Firing of events ----- */
protected synchronized void fireControlFired(ControlEvent ce) { ... }

// ...

}

/***** Assembled application *****/
public class Application {
    // ...

    public static void main(String[] args) {
        /* ----- Creating bean instances ----- */
        // Event listener 1
        SimpleController simpleController = new SimpleController();
        // Event listener 2
        Controller controller = new Controller();
        // Event source
        ControlContainer controlContainer = new ControlContainer();

        /* ----- Connecting bean instances ----- */
        controlContainer.addSimpleControlListener (simpleController);
        controlContainer.addSimpleControlListener (controller);
        // ...
    }
}

```

In this example, the instantiation of JavaBeans was done by a call to Java's new-operator instead of calling

```
java.beans.Beans.instantiate(ClassLoader cl, String beanName).
```

This is legal if the bean does not use serialized data for its initialization.

If two beans do not fit together, typically adaptor classes will be used. An adapter class implements the listener interface needed by the event source (*source*), but not implemented by the bean (*listener*) which should react to the event notifications. Instead of *listener*, an instance of the adapter class (*adapter*) is registered as a listener to *source*. If *adapter* is notified by *source*, *adapter* calls methods of *listener* according to the event notifications received.

This pattern is often used when creating user interfaces. E.g. often a mouse click on a button starts saving certain data. The bean implementing the data access will normally be unaware of special events like mouse clicks or ActionEvents because it might be used in many different environments where different events may trigger the method calls to an instance of the bean.

The composition technique introduced for JavaBeans focuses on a web of loosely coupled bean instances building a new application or applet.

2.2.1.3 Type System

In the literature, one can not find a definition for the type of a JavaBean. What can be found in [Jav97] is the following:

“In the first release of the JavaBeans architecture, each bean is a single Java object. However, in future releases of JavaBeans we plan to add support for beans that are implemented as a set of cooperating objects. One particular reason for supporting beans as sets of cooperating objects is to allow a bean to use several different classes as part of its implementation. Because the Java language only supports single implementation inheritance, any given Java object can only extend a single Java class. However, sometimes when constructing a bean it may be useful to be able to exploit several existing classes.”

Based on this text and on the following discussion, we propose to define the type of a JavaBean and subtyping between JavaBeans as done in type definition 2.2.4 and subtype definition 2.2.5 below.

A JavaBean may be used in a hand-written application or within a builder tool (IDE). In a hand-written application it is represented by its JavaBean-class and is used like a normal Java class. Only instantiation for this class may differ which should be done by a call to `java.beans.instantiate (...)` instead of only calling the new-operator. Attributes and methods of the JavaBean-class are accessed in the same way as for any other Java class.

When used in a builder tool, a BeanInfo object (see Section 2.2.1.4) is used to retrieve information on the properties, events and exposed methods of a bean. This view on the bean may differ from the class view, if the bean comes with its own BeanInfo class. A BeanInfo class may e.g. restrict the properties or methods available to clients in a tool environment. BeanInfo objects are mainly provided to enable the handling of (unknown) beans by an IDE and probably to restrict access to methods not needing expertise. The information on properties is e.g. used to fill property sheets and the information on events fired by a bean to generate code if two beans should be connected.

Although the BeanInfo view may differ from the JavaBean-class view, in any case the JavaBean-class is used to instantiate a bean. The names of the attributes and methods declared in this class are used in the client code, not the names provided by the BeanInfo class, if they are different from the JavaBean-class. Clients can call public methods which are declared by the JavaBean-class but are not exposed by the BeanInfo class etc. Furthermore every public method inherited by the JavaBean-class may also be invoked on a corresponding bean instance.

Although Sun does not explicitly provide a type definition for JavaBeans, we declare the type of a JavaBean for the reasons discussed so far as follows:

Type Definition 2.2.4 (Type of a JavaBean) *The type of a JavaBean is defined as the type of its JavaBean-class.*

Although not explicitly stated in the literature, subtyping for JavaBeans can consequently be defined as:

Subtype Definition 2.2.5 (Subtyping for JavaBeans) *The type of a JavaBean with a Java Bean-class D is a subtype of the type of a JavaBean with a JavaBean-class C , if D is a subtype of C in Java.*

2.2.1.4 Type Metadata

For the JavaBeans component model Java's reflection service as well as the `BeanInfo` interface are used to provide information about the properties, events and exposed methods of a JavaBean. By reflection, the retrieved methods can even be invoked.

Reflection: Java's reflection classes, available in the package `java.lang.reflect`, and the class `java.lang.Class` are able to retrieve type information from arbitrary Java class-files. For every inspected class, information about its attributes, constructors and methods is made available. For this purpose, `java.lang.Class` provides methods like `getConstructors()`, `getFields()`, `getMethods()`, ... which can be invoked on every `Class` object. A `Class` object can be generated from a string containing the name of the class including necessary package information (e.g. "java.lang.Integer") by a call to the method `forName` in class `Class`:

```
public static Class forName (String className) throws....
```

Another way to generate a `Class` object is by calling the `getClass()` method (declared in class `Object`) on an arbitrary object. This method returns the `Class` object corresponding to the class the object was created from.

`getMethods` returns an array of `Method` objects each representing a public method declared or inherited by the class represented by the `Class` object `getMethods` was invoked from. For every `Method` object the return type of the corresponding method can be determined as well as the types of its parameters. For this purpose, class `Method` provides the methods `getReturnType` and `getParameterTypes` with signatures

```
public Class getReturnType()          and
public Class[] getParameterTypes().
```

A `Method` object may also be used to execute the corresponding method. For this purpose, the class `Method` provides a method `invoke` with signature

```
public Object invoke (Object obj, Object[] args) throws....
```

The first parameter refers to the object the method is invoked from. The second parameter represents the arguments used for the method call. If the return type is void, a nullpointer is returned.

Knowing the name of a class, it is possible to create an object thereof by first generating a `Class` object `co` using the method `forName` from above and then calling the method `newInstance()` on `co`. `public Object newInstance() throws... is`

declared in class `Class` and may only be used, if the class corresponding to *co* has a nullary constructor².

Using these techniques, a tool can handle arbitrary, unknown classes entirely independent of user interaction. It can create an object from a class only known by the class name, it can retrieve information on the methods the object supports, it can invoke methods from this object etc.

For more information on this subject please refer to Sun's Java 2 documentation or e.g. to [HC98a, Bro97, Jav97, Szy98].

BeanInfo: The `BeanInfo` interface (declared in package `java.beans`) is used to provide JavaBean specific information, as e.g. information about the properties a bean supports, the events it emits, all methods accessible by its clients, an icon to be used as representation in a builder tool, potentially a custom editor to be used to configure the bean, and the class used as the JavaBean-class.

For every property its name, its type and its accessor methods are stored in a `PropertyDescriptor`. For emitted events, the most relevant information stored are the method objects for the methods to register and deregister listeners, the type a listener has to conform to (listener interface), and the methods belonging to the listener interface. This information is stored in an `EventSetDescriptor`. A `MethodDescriptor` is used for every method implemented by the bean and accessible by its clients. This includes the accessor methods for properties and the (de)registration methods for events. A `MethodDescriptor` contains method specific information like the name of the method, potentially the names of its parameters, and its corresponding `Method` object. This `Method` object may be used to invoke the corresponding method (see paragraph Reflection on page 26).

The most relevant declarations concerning the `BeanInfo` information are:

```
package java.beans;
public interface BeanInfo {
    ...
    PropertyDescriptor[] getPropertyDescriptors();
    EventSetDescriptor[] getEventSetDescriptors();
    MethodDescriptor[] getMethodDescriptors();
    ...
}

package java.beans;
import java.lang.reflect.*;
public class PropertyDescriptor extends FeatureDescriptor {
    ...
    public Class getPropertyType() {...}
}
```

²A nullary constructor is a constructor without parameters.

```

    public Method getReadMethod() {...}
    public Method getWriteMethod() {...}
    public Class getPropertyEditorClass() {...}
    ...
}

package java.beans;
import java.lang.reflect.*;
public class EventSetDescriptor extends FeatureDescriptor {
    ...
    public Method getAddListenerMethod() {...}
    public Method getRemoveListenerMethod() {...}
    public Class getListenerType() {...}
    public Method[] getListenerMethods() {...}
    public MethodDescriptor[] getListenerMethodDescriptors() {...}
    ...
}

package java.beans;
import java.lang.reflect.*;
public class MethodDescriptor extends FeatureDescriptor {
    ...
    public Method getMethod() {...}
    public ParameterDescriptor[] getParameterDescriptors() {...}
    ...
}

```

The process of filling in the needed BeanInfo information is as follows. Java's reflection classes are used to determine all declared and inherited public methods of the JavaBean-class. The retrieved set of publicly available methods is used to determine properties and events the JavaBean supports. Properties and events are determined based on the naming conventions for the methods used to read and write properties and for the methods to (de)register event listeners (see Section 2.2.1.1, item 3). The class `java.beans.Introspector` can be used to do this work (see Section 2.2.1.1 item 3). This class is also simply referred to as *introspector*. The `getBeanInfo`-method implemented by this class generates a BeanInfo object which can be queried e.g. for the properties, events and exposed methods of the corresponding bean.

A JavaBean may already provide BeanInfo information by itself by delivering a class, the BeanInfo class, implementing the BeanInfo interface (see also Section 2.2.1.1, item 3). The introspector incorporates the information already provided by the BeanInfo class. By reflection, it only fills in information not already available by this class (e.g. if `getPropertyDescriptors()` returns null etc.) and does not modify or extend the non-null

information retrieved by the BeanInfo class.

If the JavaBean itself does not come with a BeanInfo class, but one of its superclasses, reflection is used to fill in the BeanInfo information for the JavaBean-class and all of its superclasses up to the class providing explicit BeanInfo information. The explicit BeanInfo information is added to the already obtained BeanInfo information and is regarded as being definitive for the current class and its super classes. The introspector then proceeds as described for a bean having its own BeanInfo.

Generally, a JavaBean only comes with its own BeanInfo class, if it does not conform to the method naming conventions for properties and events or if it wants to specify additional features like e.g. an icon representing it in a builder tool or a specialized property editor to be used to change one of its properties. In addition a BeanInfo class can also be used to restrict the exposed properties, events or methods to those declared in the JavaBean-class only (that is, properties, events or methods of superclasses are not regarded) or even to a subset thereof. For more details on this subject please refer to [Bro97, Jav97, Szy98].

2.2.1.5 Compatibility / Substitutability

In this section we want to find out, whether a JavaBean can be substituted by a new version of itself or by another JavaBean without affecting clients referring to this JavaBean.

A new version of a JavaBean can be used as long as the JavaBean-class still provides all attributes and methods publicly available in former versions that is, the component interface of the JavaBean can only be augmented. The new version may come with a changed implementation of the JavaBean-class and other classes used by the JavaBean-class. The new version can even ship completely new classes not contained in older versions. The following condition summarizes, when a JavaBean can be substituted by a new version or another JavaBean.

Condition 2.2.6 (Substitution of JavaBeans) *Let the JavaBean JB with JavaBean-class JB^{class} be referred to by a client. Let JB_{new} be another JavaBean with JavaBean-class JB_{new}^{class} which is different from JB^{class} . Then JB_{new} can only be used instead of JB , if*

- *the type of JB_{new} is a subtype of the type of JB that is, JB_{new}^{class} is a subtype of JB^{class} (see subtype definition 2.2.5),*
- *a client referring to JB does not use the name of JB^{class} directly for instantiation³.*

Such a client can be an assembly tool or an IDE which, at start up, determines the set of components available from configuration files, JAR-Archives etc. By means of reflection the types of the components can be made available to a user of the tool who can in turn use this information to correctly create and access a component instance.

³The client can e.g. read the name of JB^{class} from a file instead of using the name as a constant in its program code.

2.2.2 Component Object Model (COM)

2.2.2.1 Component Model

The Component Object Model was developed by Microsoft in the early nineties. It is a binary model, independent of programming languages used to implement components and independent of the platforms the components are running on as far as the platforms provide the COM infrastructure (see [COM95, Szy98]) and the components are compiled for these platforms. Windows is the typical platform for COM, but COM is also available on e.g. Mac OS and some UNIX operating systems. We do not distinguish between COM and DCOM, an extension to COM, that allows components to interact even if they are running on different computers. The distribution is completely hidden from the developers and users of COM components by the DCOM infrastructure. While COM already supports the communication between instances of COM components running in different processes on the same machine, DCOM extends this feature to allow communication across machine boundaries. For more details on this subject please refer to [EE98, Szy98].

1. *Component:*

In COM, a component is a piece of binary software exposing its services to its clients by special interfaces. An interface groups a set of semantically related operations. For every operation, only its signature is available, no implementation. A COM interface may therefore be compared to a Java interface. COM components adhere to special rules concerning versioning, naming and implementation of their interfaces, access to their services, instantiation, lifetime management of instances and so forth. The implementation of a COM component is referred to as a *COM class* although no class constructs may be available in the programming language used for implementation (see e.g. [EE98, COM95]). That is, a COM class can be implemented by one or more classes in an OO-language or by some procedural code etc. not referring to classes at all.

An instance of a COM class is referred to as a *COM object*. A COM object need not be a pure object in the sense of an OO-language (see above), it may be realized by several objects or by other data structures.

2. *Deployment unit holding a component:*

COM classes are embedded in so-called *COM servers*. Additionally, COM servers contain *class factories* for each embedded COM class. A factory is used to create COM objects of the associated COM class. There are three different kinds of COM servers: in-process, local and remote. An in-process server runs in the same process as the client. A local server runs in a process different from the client process, but on the same machine. A remote server runs on a machine different from the one the client is running on.

On Windows, in-process servers are delivered as DLLs (dynamic link libraries) whereas local and remote servers are delivered as EXE files. Thus, the deployment unit holding a COM class, is a DLL or EXE file containing the COM server which embeds the COM class. On other operating systems corresponding file types are selected.

3. *Structure and specification of component interfaces:*

The component interface consists of a set of interfaces provided⁴ to clients and a set of so-called *outgoing* interfaces the component expects to be implemented by its clients. Clients implementing such an interface can register/subscribe at an instance of the component for notification purposes. Outgoing interfaces can be regarded as emitting special kinds of events/notifications which are identified by the methods declared in the outgoing interface. Every subscriber will be notified by calling its corresponding implementation. A COM object with outgoing interfaces is called a *connectable object*. Connectable objects are described in more detail in Section 2.2.2.2. An outgoing interface can be compared to an enabling required service interface as introduced in Section 2.1. Although not usual, outgoing interfaces can on the other hand be used by a server to ask its client for support. They then act as a required service interface (see [COM95], pp. 51).

An interface is defined as a set of related operations. Each interface is identified by a globally unique so-called *interface identifier* (IID). Interfaces are immutable that is, if a new version of an interface is created by adding or removing operations or changing semantics, this interface is regarded as being an entirely new interface. A new unique identifier has to be assigned to it.

Every component has to implement a special interface called *IUnknown*. This interface is the base interface of all other interfaces and can be used on COM objects to query for references to the other interfaces of the COM object. *IUnknown* can also be used to identify a COM object.

An interface can inherit from another interface, as e.g. from *IUnknown*. Only single inheritance is allowed for interfaces.

The set of implemented and outgoing interfaces of a component are specified using a special interface definition language (IDL) independent of a special programming language. As this IDL is Microsoft specific, it is also referred to as MIDL. For each COM interface, the IDL-declaration comprises its IID as well as a char sequence like *IUnknown*. The char sequence represents the user readable name of the interface for use in programming languages. Additionally, the IDL declaration contains declarations for all methods belonging to the interface. From this IDL definition IDL compilers generate (header) files for programming languages

⁴Such interfaces are in [COM95] denoted as *implemented* or *supported* interfaces from the point of view of the component.

enabling the use of the interface by applications. In addition, IDL compilers create proxy and stub objects for remote procedure calls.

Now we show the declaration of a COM class named `Adder` and its interfaces `IUnknown`, `ISum` and `IMessage` in IDL, where `ISum` and `IMessage` are declared in C++ and alternatively in Java⁵. The example is built from several code pieces shown in [EE98].

Example 2.2.7 (Declarations in IDL)

```

/***** Java *****/
interface ISum extends com.ms.com.IUnknown {
    int Sum(int x, int y);
}
interface IMessage extends com.ms.com.IUnknown {
    void GotMessage(int Message);
}

/***** C++ *****/
class ISum : public IUnknown {
public:
    virtual HRESULT __stdcall Sum(int x, int y, int* retval)=0;
}
class IMessage : public IUnknown {
public:
    virtual HRESULT __stdcall GotMessage(int Message)=0;
}

/***** IDL *****/
[
    object,
    uuid(00000000-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface IUnknown
{
    HRESULT QueryInterface([in] REFIID iid, [out, iid_is(iid)] void** ppv);
    ULONG AddRef(void);
    ULONG Release(void);
}

[
    object,
    uuid(10000001-0000-0000-0000-000000000001),
]
interface ISum : IUnknown
{
    HRESULT Sum([in] int x, [in] int y, [out, retval] int* retval);
}

```

⁵Unfortunately, Java is no longer supported by Microsoft.

```

[
    object,
    uuid(10000005-0000-0000-0000-000000000001),
]
interface IMessage : IUnknown
{
    HRESULT GotMessage([in] int Message);
}

[
    uuid(10000002-0000-0000-0000-000000000001),
    helpstring("Adder Class")
]
coclass Adder
{
    interface ISum;
    [source] interface IMessage;
}

```

The complete set of interfaces corresponding to a COM component that is, the component interface itself, is specified by the IDL- *coclass* declaration. In the example above the *coclass*-declaration for *Adder* specifies the component interface of the COM component *Adder*. *Adder* has one implemented interface *ISum* and one outgoing interface *IMessage*. Outgoing interfaces are marked in IDL by *[source]*.

Files containing IDL declarations are normally identified by the file extension *'idl'*.

4. *Component lookup, instantiation and access:*

Similar to interfaces, COM classes are also identified by globally unique identifiers. These identifiers are called class identifiers or short CLSIDs. These class identifiers are to be used, if a client wants to create an instance of a COM class. Such an instance can be created by a call to *CoCreateInstance*, a procedure of the COM library linked to the client code. The CLSID of the COM class to be instantiated is passed to *CoCreateInstance* as an actual parameter. The client also specifies the interface it first wants to have access to after the creation of the COM object. The client therefore also passes the corresponding IID as an actual parameter to *CoCreateInstance*. If the instantiation is successful and the COM object implements the requested interface, *CoCreateInstance* returns a pointer to this interface.

Having a pointer to this initial interface, the client can get access to the other interfaces by calling *QueryInterface*. *QueryInterface* belongs to the interface *IUnknown*.

As IUnknown is the base interface of all interfaces, every interface has to implement the corresponding operations, especially QueryInterface.

QueryInterface takes an IID as input and checks whether the COM object implements this interface. On success, it returns a pointer to an object implementing the requested interface, otherwise a null pointer is returned.

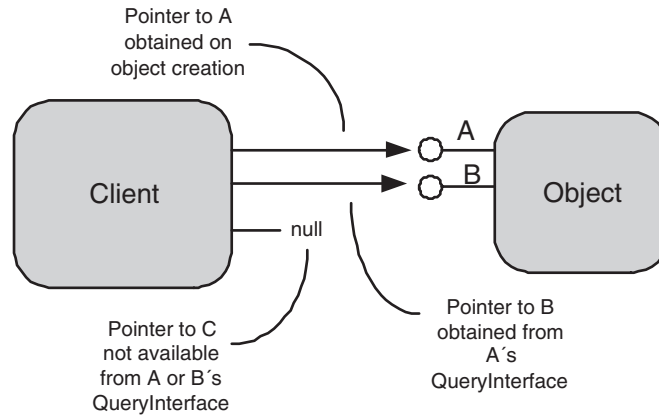


Figure 2.3: Interface Negotiation by QueryInterface

Figure 2.3 is taken from the COM specification. The lollipops sticking out of the border of the COM object denote the implemented interfaces. A and B denote the names of the interfaces.

More precisely, the COM specification says:

“When a client has access to an object, it has nothing more than a pointer through which it can access the functions in the interface, called simply an interface pointer. The pointer is opaque, meaning that it hides all aspects of internal implementation. You cannot see any details about the object such as its state information, as opposed to C++ object pointers through which a client may directly access the objects data. In COM, the client can only call functions of the interface to which it has a pointer. But instead of being a restriction, this is what allows COM to provide the efficient binary standard that enables location transparency.”

Actually, the client gets a pointer to a pointer to an array of pointers to the functions in the interface. Figure 2.4 also taken from the COM specification depicts this situation.

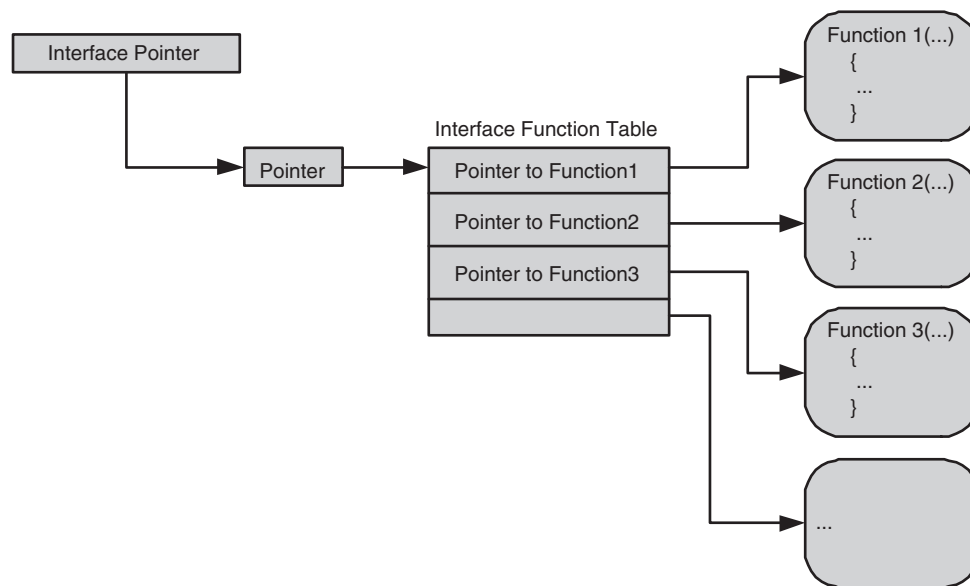


Figure 2.4: Interface Pointer

If interfaces are implemented by C++ classes, the pointer to an interface is a pointer to a C++ object. The layout of C++ objects in memory is depicted in Figure 2.5.

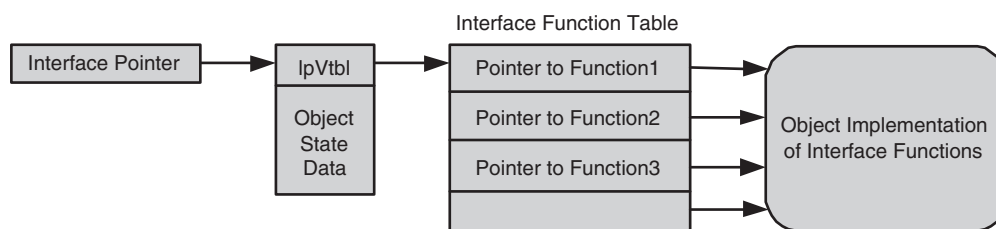


Figure 2.5: Interface Pointer as a Pointer to a C++ Object

The first entry is a pointer to the “virtual function table”, also called “vtable”. The vtable contains an entry for every virtual function (that is a function which can be overridden) of the C++ class and points to the code of the function actually to be called for the corresponding C++ object. (For more details on this subject please refer to [Str00].) As interfaces are declared as pure abstract classes declaring only virtual functions, the vtable contains pointers to all interface functions. This structure perfectly fits to the structure enforced for COM interfaces. The data of the C++ object may only be accessed from inside the COM object, not from its clients. A reference to the C++ object is implicitly passed as the first parameter to all of its functions (*this*). Therefore, the functions know the actual object data they have to use.

The same behavior is expected for other languages used to implement COM interfaces. The interface pointer is passed as this parameter to every function of the interface thereby identifying the actual “object” the function is executed on.

As a COM object may exist of many different objects, it is difficult to identify a COM object in its entirety. As every COM object has to provide the IUnknown interface, this interface is used to resolve this problem. Whenever a client queries a COM object for a reference to IUnknown, it has to return the same interface pointer. This fact can therefore be used to identify a COM object.

5. *Entities of the component interface as parameter or return value of a method:*

In COM, the accessible entities of the component interface of a component instance are its implemented interfaces. References to these interfaces may be passed as parameters to methods implemented by other COM objects, independent of whether the COM objects are distributed or not. The method *Advise* which has to be implemented for each outgoing interface to register interested ‘listeners’ can be used as an example. This method needs a reference to the IUnknown interface of the listener to be registered (pUnkSink).

```
[
    object,
    uuid ...
]
interface ... {
    HRESULT Advise([in] IUnknown* pUnkSink, [out] DWORD* pdwCookie);
    ...
}
```

Methods may also return pointers to interfaces. There are two ways to specify the type of the interface a pointer should be obtained for. Both possibilities are shown in the following example derived from examples made in [EE98].

```
...
    HRESULT GetInterfacePointer3([out] IMyCustomInterface** ppvObject);
...
    HRESULT GetInterfacePointer4([in] REFIID riid,
                                [out, iid_is(riid)] void** ppvObject);
```

The last example is the most flexible one, as it may be used for any interface. The interface requested is identified by its IID (riid, first parameter). The IDL notation *iid_is(riid)* says that the pointer to be returned has to point to the interface identified by riid.

2.2.2.2 Composition Techniques

In addition to composition by delegation or composition by only creating and accessing component instances, COM (like JavaBeans) has a mechanism to notify other COM objects of the occurrence of events. Additionally COM provides a mechanism to hierarchically compose components by *aggregation*. Both composition techniques are described in the following paragraphs.

Event notification using outgoing interfaces:

Event notification in COM is done using outgoing interfaces. This mechanism is frequently used between ActiveX controls⁶ and their containers. As already mentioned in Section 2.2.2.1, a COM-object having outgoing interfaces is called a connectable object.

A connectable object may be a source of different types of events. Each type of supported event is specified by one outgoing interface identified by its IID. For each outgoing interface the connectable object has to provide a corresponding connection point object which is able to register interested listeners. Interested listeners have to provide a *sink* object which implements the outgoing interface. A reference to the IUnknown interface of the sink is passed as a parameter to the registration method provided by the connection point object. The connection point object then asks the sink object for a reference to its implementation of the outgoing interface by a call to the sink's QueryInterface method. The connection point object stores this reference for subsequent calls of the event notification methods.

(De)Registering is done by the standard methods `Advise` and `Unadvise` of the interface `IConnectionPoint` which has to be implemented by every connection point object.

Each connectable object has to implement the interface `IConnectionPointContainer`, so that an interested listener is able to obtain a reference to a connection point object. This interface provides a method `FindConnectionPoint` which takes the IID of an outgoing interface as an input and returns a reference to a corresponding connection point object.

The following figure represents the necessary steps to establish a connection between a client which is interested in notifications of type `IMessage` and a connectable object that declares `IMessage` as an outgoing interface. 'Establishing a connection' means registering a sink object of the client as a listener to `IMessage`-notifications at the connectable object. The implementation fragment of `IConnectionPoint::Advise` shown in the gray box at the bottom of the figure belongs to the Connection Point Object for `IMessage`. A dashed arrow denotes, that the variable at the beginning of the arrow points to the interfaces at the arrowhead. A dotted arrow denotes a method call.

⁶ActiveX controls are COM components especially designed for being downloaded via internet. They substitute the formerly used, heavy weight OLE controls which had to implement a large set of mandatory interfaces. For more details please refer to [Szy98].

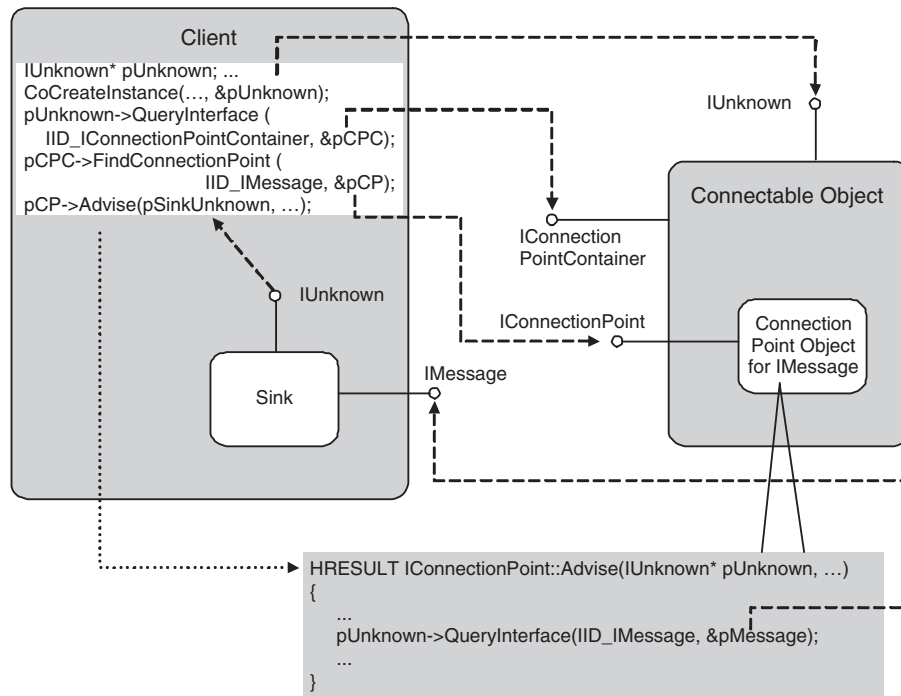


Figure 2.6: Connection between a Client and a Connectable Object

Aggregation:

Aggregation provides a means to compose COM-objects hierarchically.

Aggregation means, that a COM-object uses internal COM objects to implement a part of its functionality by directly exposing their interfaces to the environment. This behavior is shown in Figure 2.7 taken from the COM specification.

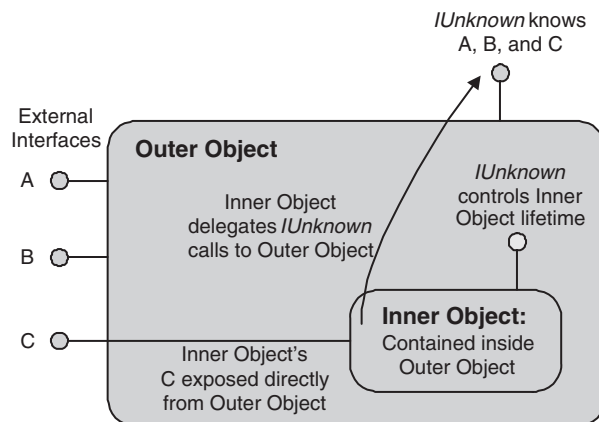


Figure 2.7: Aggregation in COM

The clients of the aggregating “outer” COM-object *Outer* do not know about the internal aggregation. They may query a reference to an interface of the aggregated

COM-object by a call to the IUnknown interface of *Outer* as if these interfaces were directly implemented by *Outer*.

The inner object can only be aggregated, if it collaborates. It has to delegate all IUnknown calls on its interfaces to the IUnknown interface of the outer object. If the outer object is queried for an interface implemented by the inner object, it delegates this query to the inner object. To avoid recursive calls, the inner object has to implement two versions of IUnknown: a delegating and a non-delegating one. The outer object must call QueryInterface on the non-delegating version of the inner's IUnknown interface.

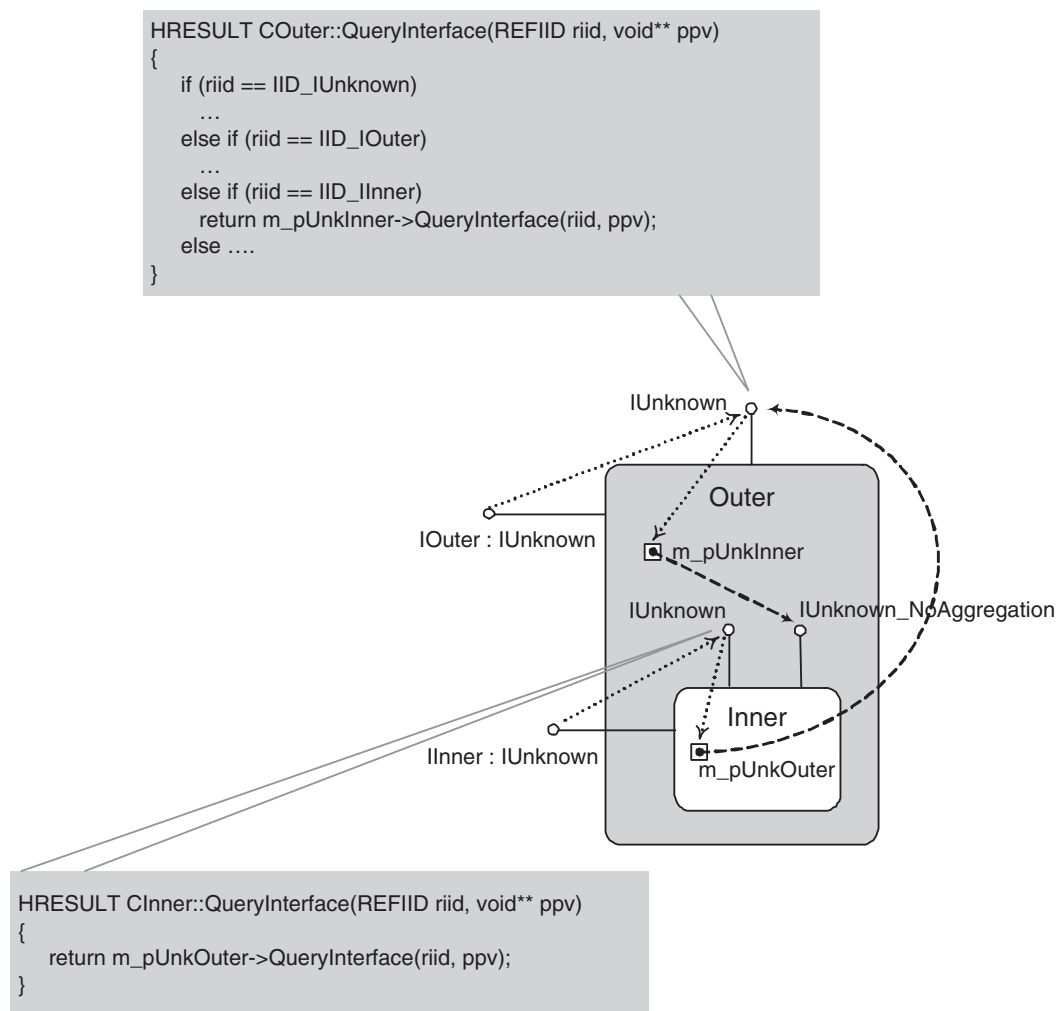


Figure 2.8: Collaboration between Outer and Inner Object in COM

In Figure 2.8 the non-delegating version of the inner's IUnknown interface is called *IUnknown_NoAggregation*. It returns references to all interfaces implemented by the inner object and returns an error, if it is queried for an interface that is not implemented. The non-delegating version of the IUnknown interface is also used for instances that are not created as a part of an outer object. The substring "pUnk" in the names of the

member variables indicates that the variables contain a pointer (p) to an IUnknown (Unk) interface.

The inner object gets a reference to the IUnknown interface of the outer object at instantiation time to enable the delegation of IUnknown calls to the outer object's IUnknown interface. The inner object stores this reference internally (see `m_pUnkOuter` in Figure 2.8) for subsequent calls to this interface. The inner object knows whether it will be aggregated or instantiated as a stand-alone object by the value of the IUnknown interface pointer of the outer object. If this value is not NULL, it is created as an aggregated object and has to return a reference to its non-delegating version of the IUnknown interface (`IUnknown.NoAggregation`) at instantiation time.

If it is created as a stand-alone object, it returns a reference to its delegating version (`IUnknown`). But instead of delegating to an outer object, it delegates to its own non-delegating version by setting the reference to the IUnknown interface of the outer object (`m_pUnkOuter`) to its own non-delegating version of IUnknown.

On the other hand, the outer object has to obtain a reference to the non-delegating IUnknown interface of the inner object. This happens, when the outer object creates the inner object by a call to `CoCreateInstance` thereby querying for the IUnknown interface of the inner object. On success, `CoCreateInstance` returns a reference to the non-delegating interface of the inner object. The outer object stores this reference internally (see `m_pUnkInner` in Figure 2.8) for subsequent calls to the IUnknown interface of the inner object. The aggregation is successfully established.

As it can be inferred from the description above, aggregation is done on the level of component instances.

One of the drawbacks of COM aggregation is that components must be aware whether they should be used in aggregates in the future. Components supporting aggregation have to prepare themselves by providing two versions of the IUnknown interface as described above and by checking at instantiation time whether they are created as a stand alone object or in the context of an aggregate. Components missing these features can never be aggregated by other components.

2.2.2.3 Type System

In this section we discuss, how types for COM interfaces and COM classes can be defined.

Interface Types: As already mentioned in Section 2.2.2.1 item 3 on page 31, COM interfaces are uniquely identified by their IIDs. In addition, they have a user readable name like `IUnknown` for use in programming languages.

As different vendors who do not know each other can introduce arbitrary COM interfaces, it can accidentally happen that two different interfaces are named equally in their IDL files. Their IIDs however are guaranteed to be different from each other. The COM specification [COM95] states:

“By contrast, COM uses globally unique identifiers (GUIDs)—128-bit integers that are virtually guaranteed to be unique in the world across space and time—to identify every interface and every object class and type. These globally unique identifiers are the same as UUIDs (Universally Unique IDs) as defined by DCE. Human-readable names are assigned only for convenience and are locally scoped. This helps insure that COM components do not accidentally connect to an object or via an interface or method, even in networks with millions of objects.”

Thus, we define the type of a COM interface as follows:

Type Definition 2.2.8 (Types for COM Interfaces) *The type of a COM interface is defined by its globally unique interface identifier (IID).*

COM interfaces can be derived from other COM interfaces like e.g. IUnknown by single interface inheritance. In the following example the IDL notation for interface inheritance is used: the derived interfaces are denoted on the left hand side of the colon, the direct base interfaces on the right hand side. The first three interfaces are used for objects to be stored persistently. IPersistStream-objects can be saved to a simple stream, IPersistFile-objects to files on disks and IPersistStorage-objects are capable to be used for structured storage⁷.

```
IPersistStream : IPersist
IPersistFile : IPersist
IPersistStorage : IPersist
IoleItemContainer : IoleContainer
```

For more details on these interfaces please refer to the MSDN-library.

Subtype relations between COM interfaces can be defined using an IDL-file. For an example please refer to Section 2.2.2.1 item 3. Type libraries (Section 2.2.2.4) can also be used to state such dependencies. Often those type libraries are generated from IDL-files.

As shown in Figure 2.9, a base interface is registered in the windows registry using the subkey BaseInterface for the IIDs of the interfaces directly derived from it:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{IID_IMoniker}
\BaseInterface={IID_IPersistStream}
```

We thus define subtyping between COM interfaces as follows:

Subtype Definition 2.2.9 (Subtypes for COM Interfaces) *A COM interface with identifier IID_1 is a subtype of a COM interface with identifier IID_2 (denoted by $IID_1 \preceq IID_2$), if IID_2 can be found in the hierarchy of direct base interfaces for IID_1 .*

⁷See MSDN-library: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/stg/stg/structured_storage_start_page.asp; 29th of April 2006.

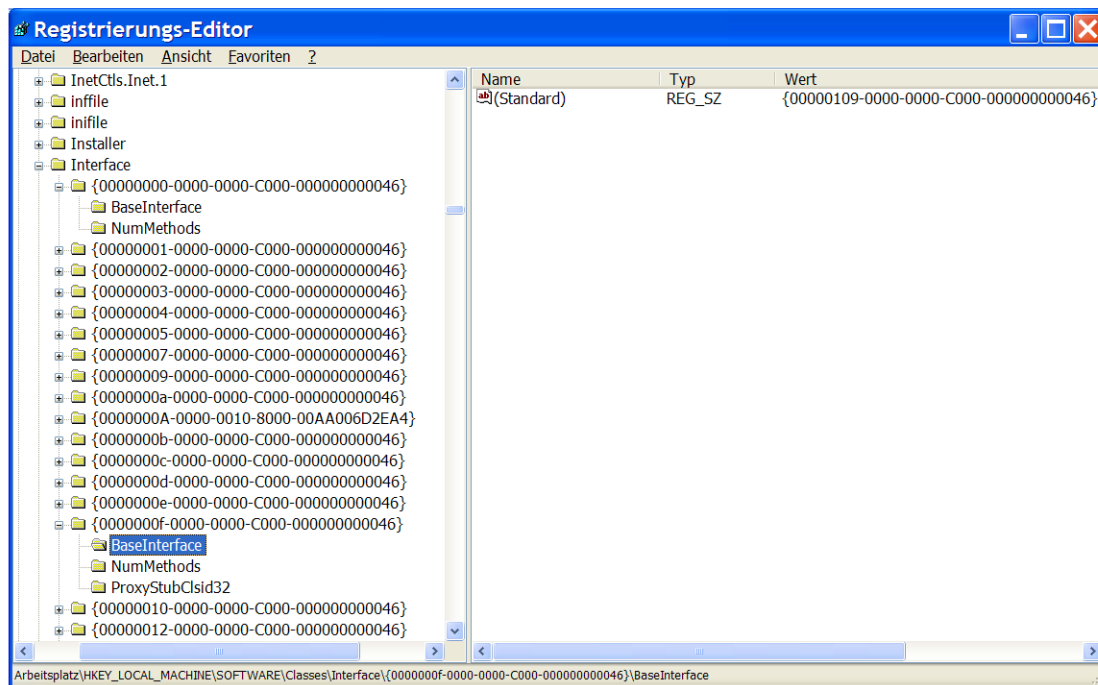


Figure 2.9: Base Interfaces in the COM Registry

Types for COM Classes: In the COM specification [COM95] the following footnote can be found contributing to what a type of a COM component could be:

“Although “class” and “type” can often be used interchangeably, in COM a type is the total signature of an object, which is the union of the interfaces that the object supports. “Class” is a particular implementation of a type, and can include certain unique implementation-specific attributes such as product name, icon, etc. ...”

The COM specification [COM95] does not explicitly introduce a subtype definition for COM classes. Instead, Section 1.4.3.1 of the COM specification describes how components may be evolved over time without forcing clients to change their code. COM classes can implement any number of new interfaces or new versions of existing interfaces as long as all formerly existing interfaces are retained. New versions of existing interfaces are treated as completely new interfaces.

Consistent with the COM specification Szyperski [Szy98] states:

“The type of a COM object is the set of interface identifiers of the interfaces it supports. A subtype is a superset of interfaces.”

Following this approach, we define types and subtypes for COM classes as follows:

Type Definition 2.2.10 (Type of a COM Class based on implemented Interfaces) *The type of a COM class is the set of interface identifiers of the interfaces it supports.*

Subtype Definition 2.2.11 (Subtyping for COM Classes based on implemented Interfaces)
Let D be a COM class with type T_D and C a COM class with type T_C . T_D is a subtype of T_C , if $T_D \supseteq T_C$.

Szyperski's type and subtype definitions perfectly fit for applications which are able to handle different kinds of COM components belonging to a common *category*.

A category in COM specifies a fixed set of interface identifiers. Categories are identified by CATIDs and are immutable like interfaces once introduced. Categories exist e.g. for Controls, Automation Objects, Document Objects, Printable Objects etc. Categories were introduced to simplify the decision whether an existing COM component implements a certain **set** of interfaces. If a COM component belongs to a certain category, this fact will be documented by an entry in the windows registry. Thus applications which can only handle COM objects implementing a certain set of interfaces identified by a category can look at the windows registry to decide, whether a COM object satisfies these requirements or not. The applications need no longer ask each COM object for every needed interface whether it implements this interface or not by calling `QueryInterface`.

Additionally, if an application identifies the components it can handle by a category, the category serves as a kind of supertype. All components being a subtype of this supertype (that is all components implementing at least the interfaces defined by the category) can be handled by the application. One component can belong to several categories.

Although Szyperski's type and subtype definitions conform to the evolution requirements for COM components, they do not consider outgoing interfaces. So we choose another definition for typing and subtyping with respect to COM components:

Type Definition 2.2.12 (Type of a COM Class) *The type T_C of a COM class C is defined by the set $I_{provided}^C$ of interface identifiers of the interfaces it implements and the set $I_{outgoing}^C$ of interface identifiers of its outgoing interfaces: $T_C = (I_{provided}^C, I_{outgoing}^C)$.*

Although outgoing interfaces are not implemented by the COM component itself, but are required to be implemented by its clients, the clients normally profit from an outgoing interface: the registered clients are notified when events, defined by the methods of the outgoing interface, occur. Thus, outgoing interfaces present in a supertype must also be available in a subtype. Otherwise a client, waiting for event notifications will not be notified any longer and thus its requirements will not be met.

On the other hand, outgoing interfaces may be used by a server to ask its client for support. Thus, outgoing interfaces can act as mandatory required service interfaces, too. Although this kind of application of outgoing interfaces is rarely used, we have to incorporate this kind of application in our subtype definition. Therefore, we must not generally allow a subtype to declare additional outgoing interfaces because they may express further requirements. Thus, our subtype definition assumes equality for the sets of outgoing interfaces.

Subtype Definition 2.2.13 (Subtyping for COM Classes) *Let C and D be two COM classes with their types T_C and T_D . T_D is a subtype of T_C , if*

- $I_{provided}^D \supseteq I_{provided}^C$ and
- $I_{outgoing}^D = I_{outgoing}^C$.

2.2.2.4 Type Metadata

COM components use *type libraries* to provide type information at runtime. For COM components, type libraries describe COM classes, COM interfaces and their dependencies. Type libraries can also be used to describe arbitrary dynamic link libraries (DLL), not dealing with COM components at all. For those DLLs, more general information is provided, as e.g. information about the interface of the DLL (called module) to its clients in terms of variables and functions. In our description, we will focus on information about COM classes and COM interfaces only. For more details please refer to [EE98] or to Microsoft's MSDN library.

Type Libraries and Interface *ITypeLib*: Every type library is identified by a so-called LIBID. If a COM class is described by a type library, this LIBID is stored in the registration database under the *TypeLib*-subkey of the CLSID key entry for the COM class. Based on this entry, the type library corresponding to a COM class can be determined.

Every entry in the type library provides the type information for one entity like e.g. a COM class, a COM interface etc. This kind of type description is referred to as *typeinfo*. The contents of the type library can be accessed by the *ITypeLib* interface. This interface provides methods to get access to the type information of an entry, to retrieve documentation information on the library itself and so forth. Its most relevant methods are the following:

```

UINT GetTypeInfoCount();

HRESULT GetTypeInfo(
    unsigned int      index,
    ITypeInfo FAR* FAR* ppTInfo
);

HRESULT GetTypeInfoOfGuid(
    REFGUID           guid,
    ITypeInfo FAR* FAR* ppTinfo
);

HRESULT GetTypeInfoType(
    unsigned int      index,
    TYPEKIND FAR* pTKind
);

```

`GetTypeInfoCount` returns the number of type descriptions stored in the type library. `GetTypeInfo` retrieves the type description stored at index `index` in the type library. `GetTypeInfoOfGuid` retrieves the type description for an entity identified by its globally unique identifier `guid`. This method may be used to retrieve information on a COM class by its CLSID or on a COM interface by its IID. `GetTypeInfoType` returns the type of a type description. Valid types are e.g. `TKIND_COCLASS` for a COM class and `TKIND_INTERFACE` for a COM interface.

Type Information for COM Classes and COM Interfaces provided by *ITypeInfo*: By the methods `GetTypeInfo` and `GetTypeInfoOfGuid` of the `ITypeLib` interface described in the previous section, one can retrieve a type description stored in the type library. Actually, one retrieves a pointer to a pointer to an interface of type `ITypeInfo`. This interface provides a lot of methods to access the corresponding type description. The most important are listed below. They are grouped into three sets of methods: one set only applicable to COM classes, another set only applicable to COM interfaces and a third set of methods applicable to both, COM classes and COM interfaces.

```

/***** COM classes only *****/

HRESULT CreateInstance(
    IUnknown FAR*   pUnkOuter,
    REFIID           riid,
    VOID FAR* FAR*  ppvObj
);

/***** COM interfaces only *****/

HRESULT GetImplTypeFlags(
    unsigned int    index,
    int*            pImplTypeFlags
);

HRESULT GetFuncDesc(
    unsigned int     index,
    FUNCDESC FAR* FAR* ppFuncDesc
);

HRESULT GetIDsOfNames(
    OLECHAR FAR* FAR* rgszNames,
    unsigned int       cNames,
    MEMBERID FAR*      pMemId
);

```

```

HRESULT Invoke(
    VOID FAR*          pvInstance,
    MEMBERID           memid,
    unsigned short      wFlags,
    DISPPARAMS FAR*    pDispParams,
    VARIANT FAR*        pVarResult,
    EXCEPINFO FAR*      pExcepInfo,
    unsigned int FAR*   puArgErr
);

/***** COM classes and interfaces *****/

HRESULT GetTypeAttr(
    TYPEATTR FAR* FAR* ppTypeAttr
);

HRESULT GetRefTypeOfImplType(
    unsigned int      index,
    HREFTYPE FAR*     pRefType
);

HRESULT GetRefTypeInfo(
    HREFTYPE          hRefType,
    ITypeInfo FAR* FAR* ppTInfo
);

```

Type Descriptions for COM Classes: The type description for a COM class contains information on the interfaces implemented by the COM class as well as on its outgoing interfaces. The type attributes retrieved by a call to `GetTypeAttr` specify e.g. the CLSID the description corresponds to and the number of implemented interfaces. The class ID can be accessed by `(*ppTypeAttr)->guid` and the number of interfaces by `(*ppTypeAttr)->cImplTypes`, where `ppTypeAttr` is a pointer to a pointer to a `TYPEATTR`-structure (see above) containing the COM class information. Readers not familiar with the declaration and use of pointers in the programming language C should refer to e.g. [Str00] for further information.

Type descriptions corresponding to the interfaces of the COM class can be retrieved by subsequent calls to `GetRefTypeOfImplType` and `GetRefTypeInfo` where the actual value for the parameter `index` is in $0 \leq \text{index} < (*ppTypeAttr)->cImplTypes$. `GetRefTypeOfImplType` first returns a handle to the requested type information. This handle is passed to `GetRefTypeInfo` to retrieve the actual type description.

Having a pointer to the `ITypeInfo` interface for the description of a COM class `CreateInstance` (see above) can be called on this pointer to create an instance of this

COM class. `CreateInstance` takes the IID of an interface as input (`riid`) and returns a pointer to a pointer to this interface in `ppvObj` on success. `pUnkOuter` must not be `NULL`, if the instance to be created is aggregated by another COM object (see Section 2.2.2.2, pp. 38).

Thus, instances of COM classes totally unknown to a tool can be created using this type information. In addition, information on the interfaces declared by the COM class can be retrieved, especially their IID's. These in turn can be used to query for pointers to those interfaces. Type libraries provide similar features as Java's or .NET's reflection services.

Type Descriptions for COM Interfaces: The type description for a COM interface provides information on its base interfaces and especially information on the methods declared by this interface. The type attributes for an interface retrieved by a call to `GetTypeAttr` specify e.g. the IID the description corresponds to (`(*ppTypeAttr)->guid`), the number of methods declared in the interface (`(*ppTypeAttr)->cFuncs`), and the number of interfaces inherited from (`(*ppTypeAttr)->cImplTypes`).

Whether an interface is an implemented or outgoing interface can be determined by a call to `GetImplTypeFlags`. A return value containing the flag `IMPLTYPEFLAG_FSOURCE` denotes an outgoing interface.

Type descriptions for the base interfaces of the current interface may be retrieved by calls to `GetRefTypeOfImplType` and `GetRefTypeInfo`.

For every method declared in the interface a special `FUNCDESC`-description can be obtained by a call to `GetFuncDesc`. This description contains information about the number and types of parameters of the method. It also provides a *member ID* by which the method is identified. This member ID has to be used when a method is executed by a call to `Invoke`. If one knows the name of a method, this member ID can also be retrieved by a call to `GetIDsOfNames`. Having the member ID of a method and a pointer to an object implementing the interface the method is a member of, one can execute this method by a call to `Invoke`. The arguments needed for the call have to be passed to `Invoke` using the parameter `pDispParams`, which internally refers to an array. Every entry in this array specifies one argument. Every argument is of type `VARIANT` essentially being a union of a restricted set of types like primitive data types, pointers to primitive data types, special arrays, and `IUnknown` etc. Each type representable by the `VARIANT` data type is identified by a constant declared in a special enumeration like e.g. `VT_BOOL`, `VT_DATE`, `VT_INT`, `VT_UNKNOWN`, `VT_BSTR` (the latter representing Unicode strings) etc.

Summary: If a COM class comes with a type library, it is possible for tools to handle this COM class even if the tools get to know about this class only at runtime. Based on the type information in the type library, a tool is able to create an instance of this class, to retrieve a pointer to every of its implemented interfaces, and to invoke methods declared by these interfaces.

2.2.2.5 Consistency / Correctness of a Composition

As described in Section 2.2.2.2, the primary composition techniques available for COM are event notifications using outgoing interfaces and COM aggregation. In the following we mention conditions which have to hold for two instances of COM components to be connected via the given composition techniques.

Condition 2.2.14 (Valid Event Connections using Outgoing Interfaces) *Let CO be a connectable object with an outgoing interface identified by IID_Event. Then only sink objects implementing IID_Event can be registered as listeners at this outgoing interface of CO.*

Using COM aggregation, a COM-object can use other COM objects to implement a part of its functionality by directly exposing their interfaces to the environment. The aggregated COM objects have to conform to special rules not explicitly summarized in the literature as we do below.

Condition 2.2.15 (Valid Part for Aggregation) *Let C be a COM class implementing the set of interfaces with interface identifiers $\{IID_1, \dots, IID_n\}$. Let $I_B = \{IID_{i_1}, \dots, IID_{i_m}\} \subseteq \{IID_1, \dots, IID_n\}$ ($i_j \in \{1, \dots, n\}$ for $1 \leq j \leq m$) be the subset of interface identifiers C expects to be implemented by another COM class applied by C using aggregation. Then a COM class B can be used for aggregation by C, if B conforms to the following rules:*

1. *B implements the interfaces belonging to I_B .*
2. *B supports aggregation. That is,*
 - *B implements a delegating and non-delegating version of IUnknown.*
 - *B delegates all IUnknown calls on its interfaces to the IUnknown interface of C with one exception: calls on the non-delegating IUnknown interface of B are **not** delegated to C (see Figure 2.8).*
 - *B returns a reference to its non-delegating version of IUnknown on instantiation, if B is instantiated as a part of an aggregate.*

2.2.2.6 Compatibility / Substitutability

At first we consider new versions of a COM class. The COM specification postulates that new versions of a COM class may only add additional interfaces. Interfaces from former versions have to be retained. As a new version retains all former interfaces, it can be used instead of the old version without effecting old clients as far as these clients don't expect outgoing interfaces from the COM class.

If an old version declares outgoing interfaces and a client uses some of these interfaces to be notified of special events, a new version also has to declare the same set of outgoing interfaces as the old version. Otherwise the client could not successfully register for event notifications provided by an outgoing interface of the old version which

is discarded in the new version. That is, the new version has to be a subtype of the old one with respect to subtype definition 2.2.13.

Now we consider the case, when a COM class can be substituted by another COM class. This situation may arise, if one wants to replace used components by components from other vendors. Every COM class to be used instead of another one has to be a subtype of the COM class to be substituted as is the case for new versions of a COM class. Additionally, the new class can only be used, if clients don't refer to the class ID for instantiation or if the new class *emulates* the old one. If one COM class with class ID `clsidOld` is emulated by another one with class ID `clsidNew`, the server for `clsidNew` is used to create an instance instead of the server for `clsidOld`. Emulation can be achieved

1. by a call to `CoTreatAsClass`, a procedure of the COM library, which takes as input the class ID to be simulated (`clsidOld`) and the simulating class ID (`clsidNew`) or
2. under windows, by directly adding a `TreatAs` key with value `clsidNew` to the CLSID entry for `clsidOld` in the registration data base.

The corresponding entry in the registration data base, which is also created by `CoTreatAsClass`, looks as follows:

```
%HKEY_CLASSES_ROOT\CLSID\{clsidOld}\TreatAs={clsidNew}
```

The following condition summarizes, when a COM class can be substituted by a new version or another COM class.

Condition 2.2.16 (Substitution of COM classes) *Let the COM class C with class ID $clsid^C$ be referred to by a client. Let C_{new} be another COM class with class ID $clsid_{new}^C$. Then C_{new} can only be used instead of C , if*

- C_{new} is a subtype of C as defined in subtype definition 2.2.13,
- one of the following conditions holds:
 - $clsid^C = clsid_{new}^C$,
 - a client referring to C does not use $clsid^C$ directly for instantiation in its code,
 - C_{new} is an emulation for C . (In this case, $clsid^C$ can be directly used for instantiation.)

2.2.3 .NET

2.2.3.1 .NET Framework

Microsoft's .NET was introduced in summer 2000. In contrast to its predecessor COM which exactly defines a component model and provides an infrastructure supporting the instantiation of COM components, their communication etc., .NET is a framework rather than a component model. Nevertheless .NET provides a lot of features supporting component based development:

- **The handling of types at runtime which are unknown to the client at development time:** .NET enables unknown types to be handled by using *type metadata* and *reflection*.
- **Platform independence:** Platform independence is supported by using an *intermediate language* (IL) the source code is compiled to and a *common language runtime* (CLR) abstracting from particular operating systems. The CLR takes on different tasks, as e.g. compilation from IL code to machine code, memory management, thread management, access to the underlying operating system or loading of classes.
- **Language independence:** Language independence is realized by compiling source code written in arbitrary .NET languages to the same IL code. The IL uses a *common type system* (CTS) all .NET languages have to adhere to.
- **Support for versioning, packaging and deployment:** *Assemblies* are the units for versioning, packaging and deployment. Assemblies essentially contain IL code, resources and type metadata. Assemblies are described in more detail in item 2 on page 53.
- **Separation of interface and implementation:** Using components, it is essential to be able to separate the interface of a component from its implementation. This requirement is addressed by introducing .NET interfaces. .NET interfaces correspond to Java interfaces. In contrast to Java, .NET enables advanced handling of interfaces. Java only provides *implicit interface implementation*. That is, a public method `m` of a class `C` implementing an interface `I` which has the same signature as a method in `I` is assumed to be its implementation. In addition to implicit interface implementation, .NET provides *explicit interface implementation*. When implementing a method explicitly, its name has to be qualified with the name of the interface declaring it. Instead of writing `void m() { ... }` in the body of `C` one would use `void I.m() { ... }`. Using explicit interface implementation for an interface `I` in a class `C`, clients can only invoke methods declared in `I` on variables explicitly typed by `I` as shown in the following code snippet:

```
I obj; obj = new C(); obj.m();
```

These variables (here `obj`) cannot be typed by `C` as would be the case when implicit interface implementation is used.
- **Support for component metadata:** Various kinds of component metadata can be provided by *attributes*. Attributes are special classes derived from `System.Attribute`. Attributes can be used in a declarative way to enrich objects like classes or class members with additional functionality without the need to implement this functionality by the object itself. The additional information provided by the attributes is added to the metadata of the objects they decorate. These additional metadata can be accessed by .NET or tools using reflection. Reading the

metadata of an object, .NET or a tool can determine its attributes and then perform the corresponding extra functionality. .NET distinguishes *standard attributes* provided by the framework and *custom attributes* which can be defined by developers.

An often used standard attribute at class level is e.g. the `Serializable`-attribute. This attribute enables the instances of a class to be serialized to a data stream without implementing this extra functionality:

```
[Serializable]
public class A {
    // ...
}
```

Attributes can even be used at the level of assemblies. Typical examples are attributes specifying the version number of the assembly, the company developing it or copyright aspects:

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyVersion("1.3.2.2")]
[assembly: AssemblyCompany("Company XYZ")]
[assembly: AssemblyCopyright("Copyright Company XYZ 2006")]
...
```

Although providing all these features, .NET itself does not give a clear definition of what a component is. Therefore different authors use different definitions even in the context of the Microsoft Developer Network Library (MSDN-Library). Juval Löwy summarizes this problem in [Löw05] as follows:

“The term component is probably one of the most overloaded and therefore most confusing terms in modern software engineering, and the .NET documentation has its fair share of inconsistency in its handling of this concept. The confusion arises in deciding where to draw the line between a class that implements some logic, the physical entity that contains it (typically a dynamic link library, or DLL) and the associated logic used to deploy and use it, including type information, security policy, and versioning information (called an assembly in .NET). In this book, a component is a .NET class.”

Other definitions of a component given in the MSDN-Library are:

- *“... In other words, a component is a compiled set of classes that support the services provided by the component. The classes expose their services through the properties, methods, and events that comprise the component’s interface.*

Simple .NET object-oriented programming involves not much more than creating a class, adding the properties, methods, and events required by the class, and including the class in different applications. A .NET component, however, is a pre-compiled class module with a .DLL (dynamically-linked library) extension."

(See [Gro02].)

- *"In the .NET Framework a component is a class which implements the interface `System.ComponentModel.IComponent` or which directly or indirectly inherits from a class implementing this interface."⁸*

(See [MSD].)

A similar definition can be found in the MSDN-library describing interface `System.ComponentModel.IComponent`:

"To be a component, a class must implement the `IComponent` interface and provide a basic constructor that requires no parameters or a single parameter of type `IContainer`."

In the context of this thesis, we shall refer to the last definition for the following reasons:

- We want to describe already *existing* component concepts and implementations in .NET. Microsoft uses this approach to deal with components which can be customized by a tool and which can be grouped by containers. Microsoft provides a lot of support for such components by classes implemented in the namespace `System.ComponentModel`. In addition to the interface `IComponent`, this namespace contains a lot of classes that are used to implement the run-time and design-time behavior of 'components' and controls. This namespace includes the core 'component' classes, base classes and interfaces for implementing attributes, type converters, binding to data sources, and licensing components⁹.
- A predefined mechanism exists to determine a component namely by checking a class to be of type `IComponent`.
- There exists a predefined way to instantiate a component namely by calling the `new`-operator.

There is no doubt that one can realize a lot of different component concepts using the features of .NET, but the definition of such components, their identification, instantiation, and access to their services are not predefined by .NET.

In the following we answer the questions concerning components, component interfaces, instantiation etc. for .NET as we have already done for JavaBeans and COM.

⁸The original text is written in German.

⁹Most of this text comes from the MSDN-library.

1. *Component:*

A component is a class which either implements the interface `System.ComponentModel.IComponent` or which directly or indirectly inherits from a class implementing this interface.

.NET provides two base implementations for `IComponent` :

`System.ComponentModel.Component` and

`System.ComponentModel.MarshalByValueComponent`. The first class is used for remotable components which can be shared between applications. Calling applications can 'directly' access such component instances by corresponding proxies. The second class can be used for "*remotable components that are marshaled by value*"¹⁰. That is, instances of such components are passed as serialized copies to the calling applications.

2. *Deployment unit holding a component:*

Components are packaged into *assemblies*. Assemblies are the units for versioning, packaging and deployment. Assemblies essentially contain IL code, resources and type metadata. Assemblies are logical units consisting of one or more physical files. Each assembly contains a manifest holding assembly metadata.

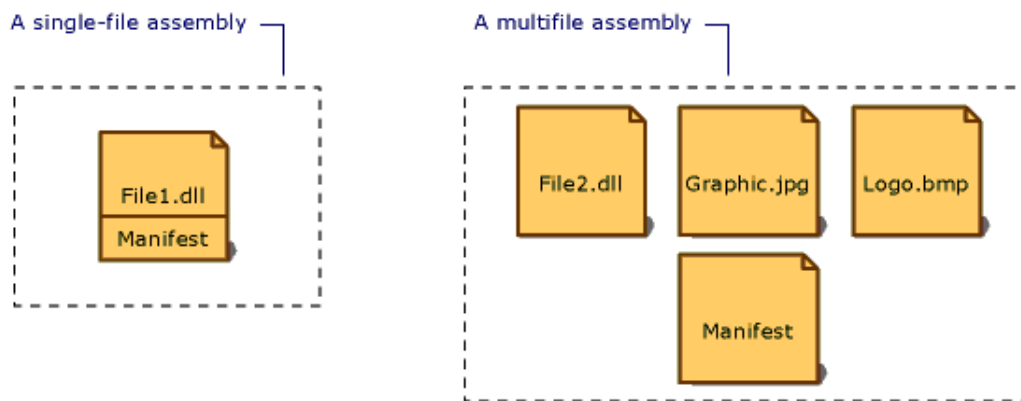


Figure 2.10: Single-file and Multifile Assembly (Source: MSDN-Library)

The manifest contains the following information:

- the assembly text name as a user friendly text string,
- the version number of the assembly,
- the public encryption key of the publisher, if the publisher generated a unique digital signature for the assembly; in this case the assembly is said to have a *strong name* (see e.g. [Löw05]),
- information on the culture or language the assembly supports,

¹⁰See MSDN-Library.

- a list of all files belonging to the assembly,
- type reference information for exported types that is, information mapping a type reference to the file containing its declaration and implementation,
- a list of other assemblies that are statically referenced by the assembly.

Although multifile assemblies are possible, typically single-file assemblies are used.

3. *Structure and specification of component interfaces:*

As components are special classes, their interface consists of the implicit interface of the class that is, its public properties, emitted events and methods. There is no extra place like an IDL-file in COM or CCM where the interface of a component is specified.

Properties: Properties in .NET encapsulate a part of the state of a class. They can be used like public fields in expressions, although their access is encapsulated by getter- and setter-methods. The following C# code example from the MSDN-Library demonstrates declaration and use of properties in .NET. The class `Employee` has a public field `numberOfEmployees` and two properties `Name` and `Counter` that can be used like public fields.

```
using System;
public class Employee {

    public static int numberOfEmployees;
    private static int counter;
    private string name;

    // A read-write instance property:
    public string Name {
        get {
            return name;
        }
        set {
            name = value;
        }
    }

    // A read-only static property:
    public static int Counter {
        get {
            return counter;
        }
    }
}
```

```

    // Constructor:
    public Employee() {
        // Calculate the employee's number:
        counter = ++counter + numberOfEmployees;
    }
}

public class MainClass {
    public static void Main() {
        Employee.numberOfEmployees = 100;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";
        Console.WriteLine("Employee number: {0}", Employee.Counter);
        Console.WriteLine("Employee name: {0}", e1.Name);
    }
}

```

Events: Classes can declare that their instances can be sources of a specific type of event. Instances of other classes able to consume such type of event can register to the event source for being notified, if this type of event occurs. The event mechanism provided by .Net is based on so-called *delegates*. A delegate type can be compared to a method type. A delegate type declaration defines a method prototype by its signature. The signature is independent of the name of the method and the names of its formal parameters. Only the return type as well as the number and types of its parameters are relevant. The name of the method occurring in the delegate type declaration denotes the delegate type which can be used for variable declarations. Such variables are able to hold one or more references to methods matching¹¹ the declared signature and are often referred to as *delegates*. Assignments to such variables in C# are done by the '+=' operator and are often referred to as subscribing/registering to the delegate. Calling the delegate results in calling all the registered methods. The following code snippet taken from [Wes02] demonstrates type and variable declaration as well as subscribing to a variable.

```

// Delegate type declaration
delegate void OnClickHandler(object eventSource);

// Variable declaration
OnClickHandler clickEvent;

// Declaration and implementation of matching methods
void button1_OnClick(object eventSource) {...}
void textbox1_OnClick(object eventSource) {...}

```

¹¹*Matching* means that the methods referred to have the same return type as well as the same number and types of parameters.

```
// Subscribing to the 'clickEvent'-variable
clickEvent += new OnClickHandler(button1_OnClick);12
clickEvent += new OnClickHandler(textbox1_OnClick);

// Invoking the delegate results in invoking
// 'button1_OnClick' and 'textbox1_OnClick' with the
// same actual parameter.
clickEvent(this);
```

Although events can be already handled by pure delegates, .NET introduces the keyword `event` to declare delegate variables to be events. This additional keyword ensures that only the publisher class¹³ can fire the event which is done by invoking the delegate. Otherwise, as delegates must be declared as public members for the (de)registration purposes described above, anyone could access the event-delegate and fire the event even if no event really takes place.

The following C# code example from the MSDN-Library demonstrates declaration and use of events in .NET. Only the code is shown dealing with the declaration of the delegate type, the declaration of the actual event, the subscription to the event, the raising/firing of the event and the declaration of data passed by the event. For the complete example please refer to the MSDN-Library.

```
namespace EventSample
{
    using System;
    using System.ComponentModel;

    // Class that contains the data for the alarm event.
    // Derives from System.EventArgs.
    public class AlarmEventArgs : EventArgs {
        // ...
    }

    // Delegate type declaration.
    //
    public delegate void AlarmEventHandler(object sender,
                                         AlarmEventArgs e);

    // Event source (publisher)
    // The Alarm class that raises the alarm event.
    //
    public class AlarmClock {
        private bool snoozePressed = false;
        private int nrings = 0;
```

¹²Subscribing can be simplified using C# 2.0 to `clickEvent += this.button1_OnClick;`

¹³The publisher class is the class declaring and firing the event.

```

// ...

// The event member that is of type AlarmEventHandler.
//
public event AlarmEventHandler Alarm;

// The protected OnAlarm method raises the event by
// invoking the delegate. The sender is always this,
// the current instance of the class.
//
protected virtual void OnAlarm(AlarmEventArgs e) {
    if (Alarm != null) {

        // Invokes the delegate.
        Alarm(this, e);
    }
}

// Actually raising the event
public void Start() {
    // ...
    AlarmEventArgs e = new AlarmEventArgs(snoozePressed, nrings);
    OnAlarm(e);
    // ...
}

}

// Event receiver (subscriber)
// The WakeMeUp class that has a method AlarmRang that handles the
// alarm event.
//
public class WakeMeUp {

    public void AlarmRang(object sender, AlarmEventArgs e) {
        // ...
    }
}

// The driver class that hooks up the event handling method of
// WakeMeUp to the alarm event of an Alarm object using a delegate.
// In a forms-based application, the driver class is the form.
//
public class AlarmDriver {
    public static void Main (string[] args) {
        // Instantiates the event receiver.

```



```

WakeMeUp w= new WakeMeUp();

// Instantiates the event source.
AlarmClock clock = new AlarmClock();

// Wires the AlarmRang method to the Alarm event.
clock.Alarm += new AlarmEventHandler(w.AlarmRang);

clock.Start();
    }
}
}

```

4. *Component lookup, instantiation and access:*

For a component which is statically referred to in an application, only its fully qualified class name has to be known. A reference to the assembly containing it is automatically created by the compiler.

If the name of the component class¹⁴ is not known at development time that is, its name is only provided at runtime e.g. by reading it from a file, one additionally has to know the assembly containing the definition and implementation of the class to be able to load the assembly dynamically. An assembly is identified by its assembly text name, its version number, its culture and the public key of the publisher if the assembly has a strong name. This information is also referred to as the *fully qualified name* of an assembly.

Instances of component classes statically referred to in an application can be simply created by using the new-operator as shown below.

```

// Declaration of the component class
class MyComponentClass : System.ComponentModel.IComponent {
    // ...
}

// Creating an instance of MyComponentClass
MyComponentClass inst = new MyComponentClass();

```

For component classes not known at development time, the assembly containing them has to be loaded first by calling the static method

```
public static Assembly Load(string assemblyString);
```

of `System.Reflection.Assembly` and passing as its parameter the fully qualified name of the assembly as string. Afterwards an instance of the component class can

¹⁴The term *component class* is sometimes used instead of *component* to stress the fact that the component consists of only one class and that a component instance is just an object.

be created calling the method

```
public object CreateInstance(string typeName);
```

of `System.Reflection.Assembly` passing as its parameter the fully qualified name of the component class as string.

An example is shown below.

```
using System;
using System.Reflection;

class MyClass {
    public static void Main()
    {
        Assembly assembly;

        // Loading an assembly with an assembly text name of 'MyAssembly',
        // a version number of 1.0.1538.0, an arbitrary culture and a
        // public key token of b77a5c561934e089c.
        assembly = Assembly.Load(
            "MyAssembly, Version=1.0.1538.0, Culture=neutral,
            PublicKeyToken=b77a5c561934e089c"
        );

        // ...

        // Creating an instance of class
        // 'MyAssemblyNamespace.MyComponentClass'
        object obj = assembly.CreateInstance(
            "MyAssemblyNamespace.MyComponentClass");
    }
}
```

Another possibility to load a type dynamically and to create an instance thereof is described in Section 2.2.3.4 on page 65.

After instantiation of a component class, the client has access to the implicit interface of the component class. Therefore it has access to all public properties, events and all publicly available methods.

5. *Entities of the component interface as parameter or return value of a method:*

As components are nothing more than special classes, their interface consists of the implicit interface of the class. This interface is not broken up into smaller entities, thus no smaller entities are to be referenced. Consequently, a reference to an instance of a component class being a pure object only, represents the only possible reference to the component interface. This reference can be passed as a parameter to methods implemented by other component classes.

2.2.3.2 Composition Techniques

Composition techniques available for .NET components in our sense are the wiring of component instances by events and the grouping of component instances into containers. Although adding components to containers is more a structuring feature than a composition technique that wires component instances together, it will be described here. This is due to the fact that the MSDN-Library stresses the ability of instances of classes of type `IComponent` to be potentially grouped by containers as an important feature.

Event notification:

The wiring of instances of .NET components by events means that component instances being capable of consuming a special type of event emitted by another component instance (the event source) subscribe to the event source for this special type of event. Then the event consumers are notified each time, the event source fires the type of event, the consumers subscribed to.

As already described in Section 2.2.3.1 item 3, in .NET event firing, event consuming, and event subscription are based on delegates. To summarize: Event types are defined as delegate types and represent method prototypes. A class is a source of a special type of event, e.g. *typeEv*, if it declares a (delegate) variable of type *typeEv* using the keyword `event`. A class is a consumer of the event type *typeEv*, if it implements a method matching the method prototype defined by *typeEv*. Instances of the consumer class can subscribe to an instance of the event source by some kind of assignment which passes a reference to the matching method to the variable representing the event. For an example please refer to Section 2.2.3.1 item 3 on page 54.

In contrast to the JavaBeans component model which supports registering to a set of events grouped by an interface, .NET supports registering to single events only.

Grouping component instances into containers:

Component instances that is, instances of classes implementing the `System.ComponentModel.IComponent` interface, can be added to an instance of a container. A container in .NET is a class implementing the interface `System.ComponentModel.IContainer`. To simplify the description, instances of containers are also referred to as containers. Containers provide the methods `Add` and `Remove` to add component instances to a container or to remove component instances from a container. New component instances are added to the end of the container's internal list of contained component instances. Clients can query a container for a list of all contained component instances. Containment in this context refers to logical containment.

When an instance of a component is added to a container, the container can create a so-called *site* object and pass it to the component instance which stores it in its `Site` property. A site object is an object of type `System.ComponentModel.ISite` and allows the container to maintain container-specific information per component as e.g. a component name. The site object holds a reference to the container as well as to the component instance and can thus be used by both to communicate with each other.

If a container additionally implements `System.ComponentModel.IComponent`, it can act as both, a container and a component and can thus itself be added to a container. This allows one to group instances of components hierarchically. Adding a component instance to a container being itself a component instance can be regarded as some kind of wiring between component instances. Nevertheless a container does not comprise the functionality of its contained component instances.

The following example is taken from the MSDN-Library. It describes a library where books can be added to and removed from. The example demonstrates the interplay of components (books), containers(library) and sites(container specific information per component, the ISBN name).

```
using System;
using System.ComponentModel;
using System.Collections;

// Implement the ISite interface.
// The ISBNSite class represents the ISBN name of the book component
class ISBNSite : ISite {
    private IComponent m_curComponent;
    private IContainer m_curContainer;
    private bool m_bDesignMode;
    private string m_ISBNCmpName;

    public ISBNSite(IContainer actvCntr, IComponent prntCmpnt) {
        m_curComponent = prntCmpnt;
        m_curContainer = actvCntr;
        m_bDesignMode = false;
        m_ISBNCmpName = null;
    }

    //Support the ISite interface.
    public virtual IComponent Component {
        get { return m_curComponent; }
    }

    public virtual IContainer Container {
        get { return m_curContainer; }
    }

    public virtual bool DesignMode {
        get { return m_bDesignMode; }
    }

    public virtual string Name {
        get { return m_ISBNCmpName; }
        set { m_ISBNCmpName = value; }
    }

    // Support the IServiceProvider interface.
```

```

    public virtual object GetService(Type serviceType) {
        //This example does not use any service object.
        return null;
    }
}

// The BookComponent class represents the book component of the
// library container. This class implements the IComponent interface.
class BookComponent : IComponent {
    public event EventHandler Disposed;
    private ISite m_curISBNSite;
    private string m_bookTitle;
    private string m_bookAuthor;

    public BookComponent(string Title, string Author) {
        m_curISBNSite = null;
        Disposed = null;
        m_bookTitle = Title;
        m_bookAuthor = Author;
    }

    public string Title {
        get { return m_bookTitle; }
    }

    public string Author {
        get { return m_bookAuthor; }
    }

    public virtual void Dispose() {
        //There is nothing to clean.
        if(Disposed != null) Disposed(this, EventArgs.Empty);
    }

    public virtual ISite Site {
        get { return m_curISBNSite; }
        set { m_curISBNSite = value; }
    }

    public override bool Equals(object cmp) {
        BookComponent cmpObj = (BookComponent)cmp;
        if(this.Title.Equals(cmpObj.Title) &&
           this.Author.Equals(cmpObj.Author))
            return true;
        return false;
    }

    public override int GetHashCode() {
        return base.GetHashCode();
    }
}

```

```
// Implement the LibraryContainer using the IContainer interface.
class LibraryContainer : IContainer {
    private ArrayList m_bookList;

    public LibraryContainer() {
        m_bookList = new ArrayList();
    }

    public virtual void Add(IComponent book) {
        //The book will be added without creation of the ISite object.
        m_bookList.Add(book);
    }

    public virtual void Add(IComponent book, string ISNDNNum) {
        for(int i =0; i < m_bookList.Count; ++i) {
            IComponent curObj = (IComponent)m_bookList[i];
            if(curObj.Site != null) {
                if(curObj.Site.Name.Equals(ISNDNNum))
                    throw new SystemException(
                        "The ISBN number already exists in the container");
            }
        }
        ISBNSite data = new ISBNSite(this, book);
        data.Name = ISNDNNum;
        book.Site = data;
        m_bookList.Add(book);
    }

    public virtual void Remove(IComponent book) {
        for(int i =0; i < m_bookList.Count; ++i) {
            if(book.Equals(m_bookList[i])) {
                m_bookList.RemoveAt(i);
                break;
            }
        }
    }

    public ComponentCollection Components {
        get {
            IComponent[] datalist = new BookComponent[m_bookList.Count];
            m_bookList.CopyTo(datalist);
            return new ComponentCollection(datalist);
        }
    }

    public virtual void Dispose() {
        for(int i =0; i < m_bookList.Count; ++i) {
            IComponent curObj = (IComponent)m_bookList[i];
            curObj.Dispose();
        }
    }
}
```

```

        m_bookList.Clear();
    }

    static void Main(string[] args) {
        LibraryContainer cntrexmpl = new LibraryContainer();
        try {
            BookComponent book1 = new BookComponent("Wizard's First Rule",
                                                    "Terry Goodkind");
            cntrexmpl.Add(book1, "0812548051");
            BookComponent book2 = new BookComponent("Stone of Tears",
                                                    "Terry Goodkind");
            cntrexmpl.Add(book2, "0812548094");
            BookComponent book3 = new BookComponent("Blood of the Fold",
                                                    "Terry Goodkind");
            cntrexmpl.Add(book3, "0812551478");
            BookComponent book4 = new BookComponent("The Soul of the Fire",
                                                    "Terry Goodkind");
            // This will generate exception because the ISBN already
            // exists in the container.
            cntrexmpl.Add(book4, "0812551478");
        }
        catch(SystemException e) {
            Console.WriteLine("Error description: " + e.Message);
        }
        ComponentCollection datalist = cntrexmpl.Components;
        IEnumerator denum = datalist.GetEnumerator();
        while(denum.MoveNext()) {
            BookComponent cmp = (BookComponent)denum.Current;
            Console.WriteLine("Book Title: " + cmp.Title);
            Console.WriteLine("Book Author: " + cmp.Author);
            Console.WriteLine("Book ISBN: " + cmp.Site.Name);
        }
    }
}

```

2.2.3.3 Type System

Due to our definition, a .NET component is nothing more than a special class, the type of a .NET component can be defined as follows.

Type Definition 2.2.17 (Type of a .NET Component) *The type of a .NET component is identical to the type of the class representing it.*

As a consequence, subtyping for .NET components is defined as:

Subtype Definition 2.2.18 (Subtyping for .NET Components) *A .NET component represented by a class D is a subtype of a .NET component represented by a class C , if D is a subtype of C in .NET.*

2.2.3.4 Type Metadata

Similar to Java, .NET supports reflection services. Using reflection, it is possible to access assembly and type metadata by program code. Reflection is available at the level of assemblies, classes/types and class members. At assembly level, the assembly attributes can be determined as well as the types contained in the assembly. At class level, one can obtain e.g. information about the implemented methods or attributes decorating the class. For a method, its signature can be obtained (class member level) and so forth.

Thus, reflection can be used to retrieve information about the fields, properties, events and methods of a .NET class and to invoke its methods.

Reflection .NET's reflection classes, available in the namespace `System.Reflection`, and class `System.Type` are able to retrieve type information from arbitrary .NET types. For every inspected class, information is made available about its fields, constructors, methods etc. For this purpose `System.Type` provides methods like `GetConstructors`, `GetFields`, `GetMethods`, `GetProperties`, `GetEvents`... which may be invoked on every `Type` object.

A `Type` object can be generated from a string containing the name of the class (e.g. "`System.Object`") by invoking the method

```
public static Type GetType (string typeName);
```

declared in class `System.Type`. `typeName`¹⁵ can be a simple type name, a type name that includes a namespace, or a complex name that includes an assembly name specification. If `typeName` includes only the name of the type, this method searches in the assembly of the calling object, then in the `mscorlib.dll` assembly. If `typeName` is fully qualified with the partial or complete assembly name¹⁶, this method searches in the specified assembly.

Another way to generate a `Type` object is by calling the method

```
public Type GetType(); (declared in class System.Object)
```

on an arbitrary object. This method returns the `Type` object corresponding to the class the object was created from.

`public MethodInfo[] GetMethods();` returns an array of `MethodInfo` objects each representing a public method declared or inherited by the class represented by the `Type` object `GetMethods` was invoked from. The class `MethodInfo` is contained in the `System.Reflection` namespace. For every `MethodInfo` object the attributes, name and return type of the corresponding method can be determined as well as the types of its parameters. For this purpose, class `System.Reflection.MethodInfo` provides the properties `Attributes`, `Name`, `ReturnType` and a method

```
public abstract ParameterInfo[] GetParameters();
```

returning an array of `ParameterInfo` objects each representing one parameter. A `ParameterInfo` object provides e.g. the name and type of the parameter as well as its attributes.

¹⁵The following description concerning 'typeName' comes from the MSDN-Library.

¹⁶The term 'complete assembly name' is used synonymously to 'fully qualified assembly name'.

Another way to obtain a `MethodInfo` object is by invoking the method `public MethodInfo GetMethod(string name, Type[] types);` on the `Type` object under consideration. This method can be called, if the method name and the types of its parameters are known.

A `MethodInfo` object may also be used to execute the corresponding method. For this purpose the class `MethodInfo` provides a method `Invoke` with signature `public object Invoke(object obj, object[] parameters);` The first parameter refers to the object the method is invoked from. The second parameter represents the arguments used for the method call.

Knowing the name of a class, it is possible to create an object thereof by first generating a `Type` object `typeObj` using the method `GetType` of class `Type` from above and then calling `System.Activator.CreateInstance(typeObj);`. `CreateInstance` has the following signature: `public static object CreateInstance(Type type);` and belongs to the class `Activator` in the namespace `System`.

Using these techniques, a tool is able to handle arbitrary, unknown classes entirely independently of user interaction. It may create an object from a class only known by the class name, it may retrieve information on the methods the object supports, it may invoke methods from this object etc.

The following example demonstrates the dynamic loading of a type, the creation of an instance of this type, the retrieval of a `MethodInfo` object for a method exposed by the type and its invocation.

```
using System;

class MyClass
{
    public static void Main(string[] arg)
    {
        try
        {
            // Get the type of a specified class.
            Type tso = Type.GetType("System.Object");

            // Get a MethodInfo object for the method 'GetType'
            // of class 'System.Object'
            System.Reflection.MethodInfo mi = tso.GetMethod("GetType");

            // Create an instance of class 'System.Object'
            object obj = System.Activator.CreateInstance(tso);

            // Invoke 'GetType' on 'obj'
            object retval = mi.Invoke(obj, null);
        }
        catch (TypeLoadException e)
```

```

    {
        Console.WriteLine(e.Message);
    }
    catch(Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
}

```

2.2.3.5 Consistency / Correctness of a Composition

As described in Section 2.2.3.2, the primary composition techniques available for .NET components are the wiring of component instances based on events and the grouping of component instances by containers.

Condition 2.2.19 (Valid Event Connections) *An event consumer and an event source can be connected via an event of type $EvtType$, if the event source declares an event variable of type $EvtType$ and the event consumer implements a method that matches the method prototype defined by $EvtType$.*

Condition 2.2.20 (Valid Container - Component Connections) *Every instance of type `System.ComponentModel.IComponent` can be added to an arbitrary container of type `System.ComponentModel.IContainer`.*

2.2.3.6 Compatibility / Substitutability

A new version of a .NET component can be used as long as the component class still provides all properties, events and methods publicly available in former versions that is, the component interface of the component can only be augmented.

Condition 2.2.21 (Substitution of .NET components) *Let the component class NC be referred to by a client. Let NC_{new} be another .NET component. Then NC_{new} can only be used instead of NC , if*

- NC_{new} is a subtype of NC in .NET,
- a client referring to the component class NC does not use the class's name directly for instantiation e.g. by using the name as a constant in its program code. Instead, the client can e.g. read the name from a file.

Chapter 3

Improvements over Existing Approaches

This chapter describes the similarities as well as the differences between the industrial component models described in Section 2.2 and identifies several of their shortcomings with respect to the component model as such (Section 3.1), their type system (Section 3.3) and the composition techniques available (Section 3.2). In addition to the composition techniques provided by the component models themselves, we shortly discuss existing tooling with respect to visual composition. For every field discussed we finally present the improvements we want to achieve over the existing approaches.

3.1 Component Models

In this section we summarize the main concepts of the existing industrial component models and discuss some of their shortcomings. Our unifying component model has to support these main concepts. To complete our discussion on existing component models, we include the Corba Component Model (CCM) and Enterprise JavaBeans (EJB). Finally we describe additional concepts we want to support and motivate them by an example.

JavaBeans and .NET: JavaBeans as well as .NET components mainly suffer from the problem that they can only declare what they provide to their clients. As they are assumed to be completely self-contained, they do not specify any requirements. Thus, they can not be used in contexts, where requirements are inevitable, as e.g. in layered architectures on a higher level. In addition, the methods they provide, that is, their contract to their clients, are not declared by a separate interface specification apart from the component class. Thus no other class than the component-class can be used for implementation. In addition, from a client's point of view, a JavaBean or .NET component only provides one single interface comprising all its functionality. There is no means to subdivide its functionality into smaller accessible pieces. Although the JavaBeans

or .NET classes can implement several different interfaces, the component model has no means to express this fact e.g. by defining the implemented interfaces as different provided service interfaces. JavaBeans provide introspection and reflection services by BeanInfo-objects and Java reflection. .NET components provide reflection services using .NET reflection.

COM: In contrast to these two models, COM supports pure interface based access to its functionality. These interfaces are declared apart from the implementation and define the contract to its clients. COM components can provide several different interfaces to their clients which can be accessed independently. In addition to interfaces the COM component implements, it can declare outgoing interfaces, which generally are used to notify registered clients which have to implement this interface. Outgoing interfaces are mainly used for notification purposes, but they can also be used by a server to obtain support from its client. Although these two applications of outgoing interfaces are fairly distinct, there is no means to distinguish them and therefore no possibility e.g. for a tool to determine, if a required connection is not established. At runtime, outgoing interfaces in COM can refuse connections, if a maximum number on connections is reached, but it is not possible to determine in advance whether an outgoing interface restricts the number of sinks which can be registered simultaneously.

The interfaces implemented as well as the outgoing interfaces of a COM component can be declared explicitly in IDL and/or in type libraries separate from its implementation. Therefore in principle different implementations of a component interface could coexist. Nevertheless, COM allows only one implementation to be used at a time. It maps every class ID to a single implementation.

For COM components introspection and reflection are supported by type libraries.

CCM: Similar to COM, CCM components allow access to their functionality through different provided interfaces, called facets. In addition, CCM allows one to declare the dependency of one component on other components in terms of required interfaces, so-called receptacles. CCM distinguishes between simplex and multiplex receptacles. Whereas simplex receptacles allow only a single connection to be established at a time, multiplex receptacles allow an arbitrary number of connections. Nevertheless, there is no possibility to declare an explicit limit on the number of connections. It is also not possible to distinguish optional from mandatory receptacles.

CCM supports connections between receptacles and fitting facets of two component instances. Connections are established on component instance level. A connection is established by registering a fitting facet of one component instances at a receptacle of another component instance. Registering is done by calling a special connect-method on the component instance declaring the receptacle. This connect-method accepts a facet with a fitting interface type. This mechanism is similar to the registering of an event listener at an event source in the JavaBeans component model. After the connection is established, the requirement expressed by the declaration of the receptacle is satisfied.

Similar to COM, the interface of a component can be declared apart from its implementation in OMG IDL, an OMG¹ specific interface definition language. The component interface specification comprises the declaration of its sets of facets and receptacles.

Reflection services are supported through interface repositories.

EJB: An EJB is a server side component which exposes its business functionality through a well defined interface, the so-called *component interface*, to its clients. Besides the component interface, a bean also has to implement a so-called *home interface* which primarily allows one to create bean instances of the desired type. These two interfaces build the contract between a client and a bean and separate the component interface from its implementation(see [EJB03]).

The component interface declaring the bean's business methods is not declared in a separate IDL-File. Instead, it is declared as a Java interface, but its name is explicitly mentioned in the deployment descriptor.

A deployment descriptor is a special XML-file containing information about a single EJB or a set of EJBs to be deployed. For every EJB the deployment descriptor contains amongst others a name under which the bean's home interface is registered in its environment, the types of the bean's home and component interfaces², the name of the bean class used as implementation etc.

An EJB can declare that it statically depends on another EJB. The used component has to be instantiated and accessed in the program code of the using component. Components refer to the used components by a logical name in their program code similar to the use of a CLSID in COM. The implementation to be used for a component referred to by a logical name is determined by entries in the deployment descriptor or by the deployer himself. The static dependencies between EJBs are also described in the deployment descriptor using the tags <ejb-ref> and <ejb-link>. This kind of dependency is a static dependency as introduced in Section 2.1 and does not support dynamic bindings between different component instances as is e.g. supported by CCM.

For introspection, the method `getEJBMetaData()` implemented by the home object returns relevant information concerning the home and component interface of an EJB. Java reflection can be used for dynamic method invocation on EJBs [DP00].

Summary: The following table summarizes the main concepts and lists for every industrial component model described so far which of the listed concepts are already fully or partly supported and which not. A concept fully supported is denoted by '+', a concept not supported by '-', and concepts partially supported by '(+)'.

¹The **Object Management Group** is a computer industry consortium founded in 1989 which developed already diverse industry standards, as e.g. CORBA, the **Common Object Request Broker Architecture**.

²The former EJB specification version 1.1 refers to the component interface as the bean's *remote interface*.

Concept	JavaBeans	COM	.NET	CCM	EJB
Communication through well-defined service interfaces which are declared apart from the implementation	–	+	–	+	+
Explicit declaration of the service interfaces which are provided	–	+	–	+	+
Explicit declaration of the service interfaces which are required	–	(+)	–	+	(+)
Declaration of dynamic dependencies	–	+	–	+	–
Declaration of the complete component interface as an entity in its own right, made apart from the component implementation	–	+	–	+	+
Declaration of a limit on the number of connections	–	–	–	(+)	–
Connections based on service interfaces	–	+	–	+	–
Reflection services	+	+	+	+	+

Table 3.1: Main Concepts of Industrial Component Models

As these industrial component models are to be integrated into a common model, at least the listed main concepts have to be supported by our common model as well as the existing connection mechanisms to resolve dynamic dependencies. Although JavaBeans and .NET components do not support service interfaces, they can easily be extended to do so.

In addition to unification, our approach should avoid the shortcomings described for the various models and support some additional useful concepts. In Section 1.1 we motivated already why it is useful to support further concepts like plugs, especially as a means to model bi-directional connections, and to distinguish between optional and mandatory required service interfaces. To further motivate these additional concepts not supported by the existing models as well as the communication through service interfaces only, we present an example from [SPH03] we will often refer to in the remainder of this thesis.

Example 3.1.1 (Wordprocessor [SPH03]) *A company develops a wordprocessor application which is very flexible, as it is composed of several components: a wordprocessor kernel, an editor, a data manager and a spell checker.*

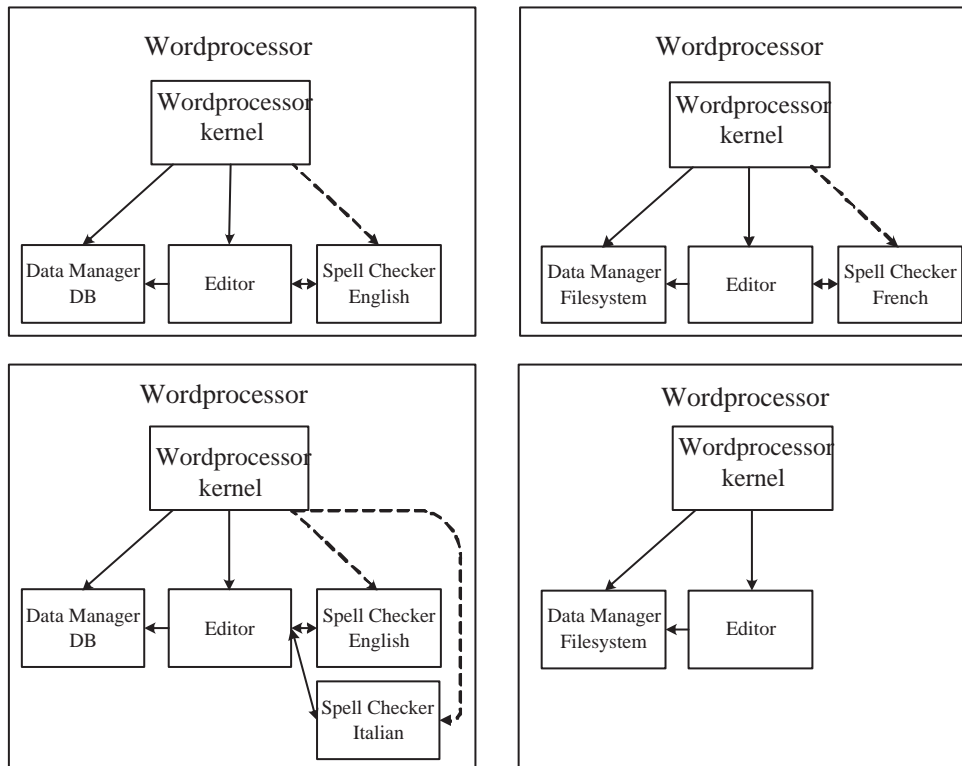


Figure 3.1: Different Configurations of a Wordprocessor Application

The wordprocessor kernel implements the overall functionality. Therefore it needs the services of the other components. Whereas the wordprocessor kernel is used in every possible configuration of a wordprocessor application, the editor, data manager and spell checker components can be exchanged by others which are more suited to the needs of a special customer. One customer for example wants to store his data into a database, another wants to use the existing filesystem. There might be customers from different countries who order different spell checkers for example in French or English. Other customers want to integrate several spell checkers, as their own customers are spread over different countries and they want to ensure that the documents/letters sent to them are free of errors. There might also be customers ordering a minimal version of the wordprocessor without any spell check functionality. Figure 3.1 shows some possible configurations of the wordprocessor application. In this figure an arrow from a component C to a component D denotes that C requires services from D . (For a distinction between solid and dashed arrows see below.)

To ensure this flexibility, the wordprocessor kernel must declare which interfaces it requires from the editor, data manager and spell checker components. Then these components may be

substituted by others later on, if the new components provide at least the interfaces required by the wordprocessor kernel. On the other hand, the data manager, editor and spell checker components must declare which interfaces they provide so that the entity connecting an instance of the wordprocessor kernel and an instance of a needed component is able to decide, whether these two component instances fit together. In this context “connecting” an instance of a component C (C_{inst}) to an instance of a component D (D_{inst}) means that C_{inst} gets a means to access the needed interfaces of D_{inst} .

Some interfaces allow multiple connections as the interface to the spell checking component for example. This means, the wordprocessor kernel can get access to the interfaces of more than one spell checker (see configuration 3 in Figure 3.1). There are connections, which are **mandatory** (denoted by solid arrows) and others which are **optional** (denoted by dashed arrows). To provide its minimal functionality, an instance of the wordprocessor kernel needs a connection to an instance of a data manager and to an instance of an editor. Without these connections, an instance of the wordprocessor kernel is not able to provide its services. So these connections are mandatory. On the other hand, the connection to an instance of a spell checker is optional. An instance of the wordprocessor kernel can be used without any spell checking functionality.

If a spell checker is used, there exists a close interconnection between the instances of the spell checker and the editor. Both component instances need services of one another. The spell checker instance must get access to the current page and the editor instance must get the misspelled words from the checker instance. This bi-directional connection must be established correctly.

3.2 Composition Techniques

In this section we summarize the existing composition techniques described in Section 2.2 and their shortcomings. In addition to the composition techniques available for the industrial component models we discuss visual assembly using builder tools, as simplifying composition by visual support through tools was one of the goals of this thesis. Other related areas as composition languages, component-oriented programming languages and architecture description languages are discussed in Section 6.

3.2.1 Industrial Component Models

As can be inferred from the composition techniques discussed for the different component models in Section 2.2, most of the current industrial component models only provide a flat component model. That is, there are predefined means to connect two component instances e.g. by events or services, but there is no possibility to define new components from existing ones by techniques other than using normal programming languages. Often even interconnections based on interfaces are not supported as is essential for many compositions like the one described in example 3.1.1. All industrial

component models lack the possibility to establish bi-directional connections by a suitable concept. With respect to the different component models we can state the following:

The JavaBeans component model e.g. only provides a predefined means to connect two component instances using event connections. Besides using program code the model does not provide any other means to describe a set of interconnected component instances making up an application or a new JavaBean. The same holds for our .NET components.

CCM [COR02] already provides a means to describe a set of interconnected component instances and their mapping to hosts and processes by a so-called assembly descriptor. Nevertheless, this kind of description does not support the building of new components with a dedicated interface. Interface based connections are supported by wiring receptacles of one component instance to fitting facets of other component instances.

For EJBs [EJB03] it is possible to define a set of cooperating EJBs. This is done in a special section of the deployment descriptor. It can be defined which implementations to use for EJBs referred to by other EJBs of the set. As in the case of CCM this is no means to define new components with a dedicated interface. The used components already have to be instantiated and accessed in the program code of the using components. The only advantage using this approach is the possibility of a late binding. Components refer to the used components only by a logical name in their program code similar to the use of a CLSID in COM. The implementation to be used for a component referred to by a logical name is determined by entries in the deployment descriptor or by the deployer himself.

In contrast to the above mentioned flat industrial component models, COM offers a means to hierarchically compose components by aggregation yielding new components (see page 38). Unfortunately, there are several drawbacks in this approach: One is that components must be aware whether they should be used in aggregates in the future. If components want to be able to be aggregated, they have to provide special additional features such as two versions of the IUnknown interface: a delegating and a non-delegating one. Additionally, they have to be aware whether they are instantiated as a stand-alone or an aggregated instance. Components missing these features can not be aggregated later on. Another drawback is that the means for hierarchical composition provided by COM are only targeted to programmers. They have to use existing programming languages to compose components, which do not support component composition concepts in a first class manner. Connections based on interfaces can be established using outgoing interfaces.

We want to improve the composition techniques available by supporting connections via service interfaces and plugs and by a suitable means to compose components hierarchically by simple techniques. One should be able to apply these composition techniques also to components of the industrial component models.

3.2.2 Visual Assembly

Several commercial IDEs like Borland's JBuilder and Delphi, IBM's Visual Age, Microsoft's Visual Basic, Visual C++ and Visual Studio 2005 already support some kind of visual assembly especially in the context of GUI elements. From the visual representation of the assembly, the tool generates program code which can be adapted by the developer. Thus, this approach needs still skilled developers. It is too complicated especially for less experienced users as assemblers. They still have to write at least glue code to connect component instances. On the other hand, only special kinds of interconnections are supported by such tools, e.g. event connections.

Besides commercial IDEs some test environments exist as e.g. Sun's BeanBox or Bean Builder [Bea]. Those tools mainly serve to test components and to demonstrate the assembly mechanisms supported by the underlying component model as well as the adapter mechanisms supported. The components can be tested for their event behavior and for a proper behavior when using property sheets. An assembly can be stored to disk and reloaded. BeanBox and Bean Builder both only support event connections as does the Component Work Bench [Obe01, OG02], CWB for short. The CWB is a prototype of an assembly tool which allows instances of components belonging to different component models to be assembled in the same *scenario*. A scenario is a loose assembly of component instances interconnected by events. No application can be built from a scenario nor new components built. As CWB, BeanBox and Bean Builder do not support the creation of new components from the assembled ones.

All the tools mentioned so far do not support advanced features like visual connections based on interfaces or the detection of components relying on other components including the case that two components rely on one another (bi-directional connection). If the assembler does not know about these dependencies and needed components are not installed or instances are not connected properly, a runtime error will occur later on or the application will not behave as intended. Therefore, assembly tools must be able to check a configuration for consistency and to simplify for example bi-directional connections.

If visual assembly is only based on event connections, used components are assumed to be completely independent of one another. Some of them are able to emit events, others are able to react on such an event or otherwise glue code / adapters may be generated reacting on the emitted event and in turn calling a desired method of a target. This approach is especially suited for GUI components. But there are many scenarios where components rely on other components and event connections are not appropriate as in example 3.1.1 on page 72.

WREN [LR01] is a prototype of an assembly tool which assembles components based on JavaBeans. But in addition to JavaBeans as introduced by Sun, these beans have provided and required service interfaces called *ports*. WREN addresses some requirements on tools which are especially targeted to assembly tools. WREN provides a type and an instance view for components, it allows one to search component pools for a set of components which fit to a user query and to detect components in an assembly with open

requirements. Connections in WREN are based on interfaces. Interface connections are declared on type level, not on instance level. New compositions yield applications. Unfortunately, new components can not be built from assembled ones.

Although WREN already addresses requirements, the other tools do not address, there are still a lot of requirements on an assembly tool not tackled. Lürer and van der Hoek [LvdH02] summarize requirements on the composition process and give an overview of some further tools. For them an ideal composition process ends up in a composed application and is divided into different phases as searching for components matching a given requirement, selecting components from the set of components found, adapting components, composing them, checking a composition for consistency, and executing a composed application either in the context of the composition environment or as stand-alone application. Current tools only partly support this process.

In the following we summarize our main requirements from Section 1.1 on the support an assembly tool should provide to its users. Except adaptation, all phases of the composition process as described in [LvdH02] are tackled. In addition, we demand that our tool also supports the building of new components by means of composition, the deployment of newly created components and support for reconfiguration of existing compositions. We support connections based on events, service interfaces and plugs.

It follows the summary on the support our tool should provide to its users. The tool should at least

- provide the user with a tool box of components he can choose from and a composition window where the user can place instances of selected components for configuration,
- support the selection of components suitable to a requirement specified by the user or a component already selected for composition,
- allow a user to connect component instances residing in the composition window visually,
- support interconnections based on events components emit or consume, interconnections based on services the components provide or require, as well as interconnections based on plugs,
- support the creation of new components by means of hierarchical composition,
- support the creation of new applications,
- provide consistency checking for newly created components, and applications thereby distinguishing the handling of optional and mandatory required services,
- support the deployment of newly created components,
- support the reconfiguration of existing compositions.

3.3 Type Systems for Components

In the following we discuss the available type and subtype definitions for the industrial component models described in Section 2.2 and summarize the diverse conditions one or more component models impose on subtypes. We also shortly discuss the Corba Component Model and Enterprise JavaBeans. The type and subtype definitions available will be checked for their suitability with respect to consistency checks and substitutability of components. We shortly discuss the conditions our subtype definition should satisfy.

JavaBeans: For type and subtype definitions for JavaBeans we refer to our definitions on page 26. Thus the type of a JavaBean is equal to the type of the corresponding JavaBean-class. This class defines the events emitted by its registration methods as well as the types of listeners which may register for notifications. This information is used by compilers and assembly tools for connections based on events.

JavaBeans having a JavaBean-class D extending another JavaBean-class C are subtypes of the JavaBean with JavaBean-class C . Although classes are not the same as interfaces in our sense because classes additionally carry implementation and probably public fields, we focus on the implicit interface of the class as if it were an explicitly, separately defined interface. Then a subtype provides at least the same interface to its clients as its supertype. That is, all properties, events and methods available in the supertype are also available in the subtype, especially the methods to register interested listeners. Thus, event connections which can be established on an instance of a supertype can also be established on an instance of a subtype.

Nevertheless, as already stated in Section 2.2.1.5 a JavaBean being a subtype can only be used instead of the original JavaBean in contexts, where the name of the JavaBean-class is not directly used in the program code/assembly code for instantiation. Thus, this kind of substitutability is not independent of the program/configuration in which the JavaBean occurs.

As JavaBeans do not have a means to express dependencies between components in terms of interfaces, their type and subtype definitions can not consider required service interfaces and bi-directional connections based on interfaces.

COM: Generally, known type and subtype definitions for COM components only refer to their implemented interfaces not to their outgoing interfaces (see type definition 2.2.10 and subtype definition 2.2.11 on page 42). In this case, not knowing what types of outgoing interfaces exist, it can not be checked, whether a sink object can be registered at an outgoing interface for notification purposes. To be able to do this check, our type and subtype definitions 2.2.13 including outgoing interfaces must be used.

The generally known subtype definition for COM components only assures that a subtype provides at least all service interfaces to its clients its supertype provides. Although this definition ensures that, if the supertype implemented `IConnectionPoint`

Container, the subtype also implements this interface, it does not ensure that the subtype has at least the same set of outgoing interfaces as its supertype. This leads to problems if clients expect the outgoing interfaces of the supertype to be available, as they want to be notified via these outgoing interfaces (see Section 2.2.2.3). When such client code should not become invalid, subtype definition 2.2.13 has to hold. But even this definition can bear problems in the case of outgoing interfaces. If an outgoing interface internally restricts the number of connections to it by an upper limit, a subtype could reduce this upper limit and thus not all connections which could be formerly established can be established when using a subtype instead of the supertype.

Subtyping for service interfaces is not supported. A service interface implemented or declared as an outgoing one by both, sub- and supertype components, has exactly to provide the same set of operations. This set is defined by the IID identifying the interface.

As for JavaBeans, all type and subtype definitions do not consider bi-directional connections.

.NET: As a .NET component, as considered in this thesis, is nothing more than a special class, its type is the type defined by the corresponding .NET class and subtyping is reduced to subclassing (see type and subtype definitions 2.2.17). Thus the discussion for .NET components is the same as for JavaBeans.

CCM: In OMG IDL, an interface definition language used for Corba objects and components, component types can be defined [COR02]. A component type includes amongst others the set of facets (provided services) as well as the set of receptacles (required services) declared for this component. Thus, knowing the interfaces typing the facets and receptacles, they can be compared to decide whether a facet of one component instance can be connected to a certain receptacle of another component instance.

In OMG IDL it is even possible to specify that a component B inherits from a component A . From this definition it can be inferred that B has at least all facets and receptacles of exactly the same types as A . Thus all former clients of A can refer to B instead. The facets they expect are available and declared connections can still be established. Thus former configurations can be retained. The only limitations concern the types for interfaces belonging to both, sub- and supertype components. They have exactly to be the same. This holds for facets as well as for receptacles. Multiplex receptacles allowing multiple connections retain this ability when inherited. That is, if a receptacle of a supertype is a multiplex receptacle, the corresponding receptacle of the subtype has also to be a multiplex receptacle.

One severe problem with the kind of subtyping for CCM components is that subtypes may define additional receptacles. Thus a component of a subtype can require more interfaces than the supertype. These additional requirements can probably not be satisfied by the environment of the component.

As in the case of the other component models, the type and subtype definitions do not consider bi-directional connections.

EJBs: The EJB component model provides a separation between the client view of a bean (as presented by its home and component interfaces) and the enterprise bean class (which provides the implementation of the client view). Nevertheless, the EJB specification [EJB03] does not give an explicit definition for the type of an EJB nor for subtyping between EJBs. An implicit definition can be found in the following text referring to future releases:

“The current EJB specification does not specify the concept of component inheritance. There are complex issues that would have to be addressed in order to define component inheritance (for example, the issue of how the primary key of the derived class relates to the primary key of the parent class, and how component inheritance affects the parent components persistence).

However, the Bean Provider can take advantage of the Java language support for inheritance as follows:

- *Interface inheritance. It is possible to use the Java language interface inheritance mechanism for inheritance of the home and component interfaces. A component may derive its home and component interfaces from some “parent” home and component interfaces; the component then can be used anywhere where a component with the parent interfaces is expected. This is a Java language feature, and its use is transparent to the EJB Container.*
- *Implementation class inheritance. It is possible to take advantage of the Java class implementation inheritance mechanism for the enterprise bean class. For example, the class `CheckingAccountBean` class can extend the `AccountBean` class to inherit the implementation of the business methods.”*

Therefore we define the type of an EJB as the set of its home and component interface and define an EJB B to be a subtype of an EJB A , if the home and component interfaces of B are equal to or subtypes of the corresponding interfaces of A . This ensures that all former clients of A can refer to B instead.

Note that this type and subtype definitions do not consider the static dependencies on other enterprise beans (called *EJB references*) which is the harder part in subtype definitions.

Concerning EJB references the EJB specification only tells when an EJB can be used as a target bean for an EJB reference:

“...the home and component interfaces of the target enterprise bean must be Java type-compatible with the interfaces declared in the EJB reference.”

Summary: The following table summarizes the conditions one or more component models impose on subtypes. For COM we refer to the generally excepted type and subtype definitions 2.2.10 given on page 42. A condition being met is denoted by '+', a condition not met by '-'. Conditions partially met are denoted by '(+)'. E.g. condition 2 is only partially met if corresponding service interface types are restricted to be equal. If conditions are not met because the corresponding concepts as e.g. required service interfaces do not even belong to the component type, this is denoted by '-'.

Condition	JavaBeans	COM	.NET	CCM	EJB
A subtype provides at least all service interfaces its supertype provides.	(+)	+	(+)	+	+
For every service interface already provided by the supertype the type of this interface is equal to or a subtype of the type of the former interface.	(+)	(+)	(+)	(+)	+
A subtype declares all service interfaces as required ones that its supertype declared to be required ones.	-	-	-	+	-
For every required service interface already declared by the supertype, the type of this interface has to be equal to or a supertype of the type of the former interface.	-	-	-	(+)	-
If a required service interface R of the supertype allows more than one connection, the subtype allows at least the same number of connections.	-	-	-	+	-

Table 3.2: Conditions on Sybtypes

Only CCM explicitly supports the concept of a component type and subtyping between components. For the other component models, type and subtype definitions could only be inferred from implicit statements of their specifications or discussions in the literature.

Similar to CCM we will define precisely what a component type is and which subtyping rules hold. Our type and subtype definitions must include provided and required services as well as our additional concept, plugs. Mandatory and optional required services have to be distinguished, due to their distinct nature. Our type system should be more flexible in that it also should allow sub- or supertypes for service interfaces and that component subtyping is based on structural equivalence instead of an explicit definition as in CCM. In contrast to CCM, a subtype must not have more required services than its supertype due to the problems already described in the context of CCM.

Our subtype definition should ensure that a component A of type T_A can be replaced by a component B of type T_B without effecting any existing configuration formerly referring to A , if T_B is a subtype of T_A .

In this context, a configuration may describe an assembly, a new component or an application which is built from interconnected component instances. Such a configuration can e.g. consist of program code or some kind of assembly description or it can be an assembly package for CCM components [COR02] etc.

If an existing configuration must not be changed, no additional interconnections can be declared, no additional component instances added, no component instances removed nor existing connections removed. That is, the new component must have the same behavior as the old one with respect to all former existing provided functionality as well as all former existing connections.

Chapter 4

Our Approach

This chapter presents the main contributions of this thesis already listed in Chapter 1:

- Contribution 1: Unifying component model with plugs (Section 4.1)
- Contribution 2: Hierarchical composition (Section 4.2)
- Contribution 3: Type system for our components (Section 4.3)
- Contribution 4: Support for component substitution (Section 4.8)
- Contribution 5: Features supporting visual composition (Section 4.11)

This chapter mainly describes the contributions we make to achieve the desired improvements discussed in Chapter 3. In the following, every contribution is briefly sketched. A detailed presentation is given in the corresponding sections of this chapter. Furthermore, in Section 4.10 we will show how industrial components can be integrated into our component model, one of the main goals of this thesis.

Component model: We introduce a hierarchical component model with the following characteristics:

- Component instances communicate to the outer world only through their component interfaces. The component interface determines the functionality of the component which can be accessed by other components. The overall functionality is divided into a set of smaller functional units which can be accessed individually. Every functional unit is represented by a service interface grouping the operations implementing this functionality. The overall functionality provided by the component is therefore determined by the union of all of its provided service interfaces.

A component can call operations belonging to service interfaces of other components which is referred to as *service communication*. This kind of communication is unidirectional.

- A component may need functionality from other components to work properly. If the component can not resolve this dependency by itself as it does not know in advance the components which will be used to provide the needed functionality, the dependency is explicitly stated in the component interface in terms of required service interfaces. If a component declaring service interfaces as required ones is used as part of an application, these open references have to be resolved in the context of the application. That is, the application must contain another component providing the needed functionality and this component must be made known to the component requiring this functionality. This process is referred to as *connection*.
- If two components need mutual access to some of their provided functional units, a bi-directional communication is needed. Especially for this purpose a new concept called *plug* is introduced which allows a programmer to express such mutual dependencies in the component interface. A plug allows a programmer to group the service interfaces of both components involved in the bi-directional communication. Plugs are units of interconnection and as such they simplify the establishment of bi-directional connections. Other mechanisms used for bi-directional communication as callbacks or asynchronous communication via special protocols do not have a programming language counterpart enclosing both directions of communication and acting as a unit of interconnection between two parties.

Hierarchical composition: The component model comes with a simple, high level language to specify component interfaces and component implementations. Each component implementation explicitly specifies, which component interface it implements. Component implementations describe atomic components as well as hierarchically composed components. Using this language, several component instances can be aggregated and wired together to build a new, higher level component exporting dedicated service interfaces and plugs of its constituents. A component refers to one of its constituents only by a name and the component interface type of the constituent. The language thus adheres to the principle of interface based programming [SM05, Pat00] in the context of components. For situations in which it is necessary to determine a certain implementation to be used for an individual constituent of a composite component or for all constituents typed by the same component interface, the language gives a convenient support. It allows a programmer to bind a special component implementation to a component interface type or to an individual constituent. If such a binding is restricted to an individual constituent, the specified component is used to instantiate the corresponding constituent only. If the specified component implementation C is bound to a certain component interface CI , C is used to instantiate all constituents of the composite component which are typed CI . To the best of our knowledge, this feature does

not exist in other composition languages or component oriented languages. In addition to other languages, our language also supports interconnections and exports based on plugs.

Integration of industrial components: Industrial components can be integrated into our component model as atomic components. The integration is achieved by providing a component interface specification and a component implementation for atomic components written in our language. Such an atomic component implementation essentially consists of a reference to the component of the industrial component model. Higher level composite components can be built from atomic components and other composite components using our language.

Type system: A type system for components including plugs and constraints on interconnections is introduced. Such a type system is needed for various reasons. First of all, component implementations for composite components use field declarations for their constituents. A constituent must be typed somehow to know the service interfaces which can be exported and to know which connections to required service interfaces have to be established for the constituent. Secondly, types are needed to decide, whether two component instances can be connected to each other via an interface or plug. Thirdly, types and subtype relations are needed to decide, whether one component can be substituted by another one without breaking the contract for existing composite components internally referring to the component to be substituted.

Besides the integration of plugs and special constraints, the type system differs from existing approaches in the handling of services a component needs from other components to fulfill its task. Our definition ensures that a component already referred to in composites can be substituted by a compatible one without affecting these composites.

Support for component substitution: Composed components can use strict interface based programming when declaring their constituents. That is, a composed component is not forced to specify a component implementation for its constituents. Thus, an explicit binding may be omitted. In this case, the choice of a suitable component implementation is left to the runtime system. Such an implementation can be substituted by another one, if the new implementation has the same component interface. A new component is even allowed to implement a subtype of this component interface type as will be shown in Section 4.8. This late binding supports the exchange of component implementations throughout the life cycle of a component.

Support for visual composition: In addition to the features already discussed in Section 1.1 and the features presented when describing our prototypes of assembly tools (Section 5), visual composition is supported by some algorithms which determine fitting plugs and mappings between two plugs. These algorithms help us to establish plug connections without any user interaction.

4.1 The Unifying Component Model (UCM)

In this section we describe our Unifying Component Model, UCM for short. We focus on the elements of the component interface used for inter component communication and on the built-in mechanisms to establish interconnections between component instances. Such built-in mechanisms are needed, if component instances shall be connected by a third party without changing the code of the components involved. Thus, these mechanisms ensure that components can be composed/connected as black boxes by a third party.

4.1.1 Basic Component Model

4.1.1.1 General Basic Concepts

In this section we describe the main ideas of UCM. We introduce the fundamental notation of a *service* and a *plug*, the basis for all interconnections between component instances. Furthermore, we describe the process of connecting one component instance to another one.

We start with the description of a component instance.

A *component instance* consists of a set of cooperating objects implementing the functionality provided by the component instance. A component instance offers its services through *service objects*. Clients can only communicate with a component instance through these service objects. A component instance can provide more than one service. Every *provided service* has a name and a type. The name has to be used by clients to get access to the corresponding service object. The type declares the set of operations a client can invoke on the service object. The type only declares method signatures. It does not carry any implementation. To stress this fact and the fact that the type constitutes an interface between a client and the component instance, the type of a service is referred to as *service interface type*. A service object is not forced to implement only one service interface type. Instead, it can implement several service interface types. One service object can even be the sole object making up a component instance. On the other hand, a component instance can support several services of the same type which refer to different service objects implementing the same service interface type.

One service object can be referenced by several clients.

A component instance need not be fully self-contained that is, to fulfil its task, it needs to call operations on service objects of other component instances. On the other hand, component instances can call operations of other component instances (registered listeners) to notify them of the occurrence of events they are interested in. Therefore component instances may themselves be clients of other component instances. The services a component instance accesses from other component instances are its *required services*. These are divided into *mandatory* and *optional* required services. Whereas a component instance can not live without having access to a service object implementing the operations of a mandatory required service, it need not have a reference to a service

object implementing the operations of an optional required service which is often used for notification purposes (see above).

Example 4.1.1 (Collection of Customer Data) In Figure 4.1 component instances and objects are denoted by rectangles with rounded edges. A provided service is denoted by a lollipop (o-), a service object as an object with lollipops sticking out of its border. A lollipop is adorned with the name of the provided service. The corresponding service object contains its name followed by a colon followed by the service interface type(s) of the service(s) implemented by the service object. The dotted rectangles with rounded edges denote proxies for services needed from other component instances (required services). These proxies hold references to service objects on which the needed operations may be called. A proxy holding a reference to a service object is denoted by an outgoing line ending in a semicircle, which touches the lollipop of the service object referenced (-(O-)).

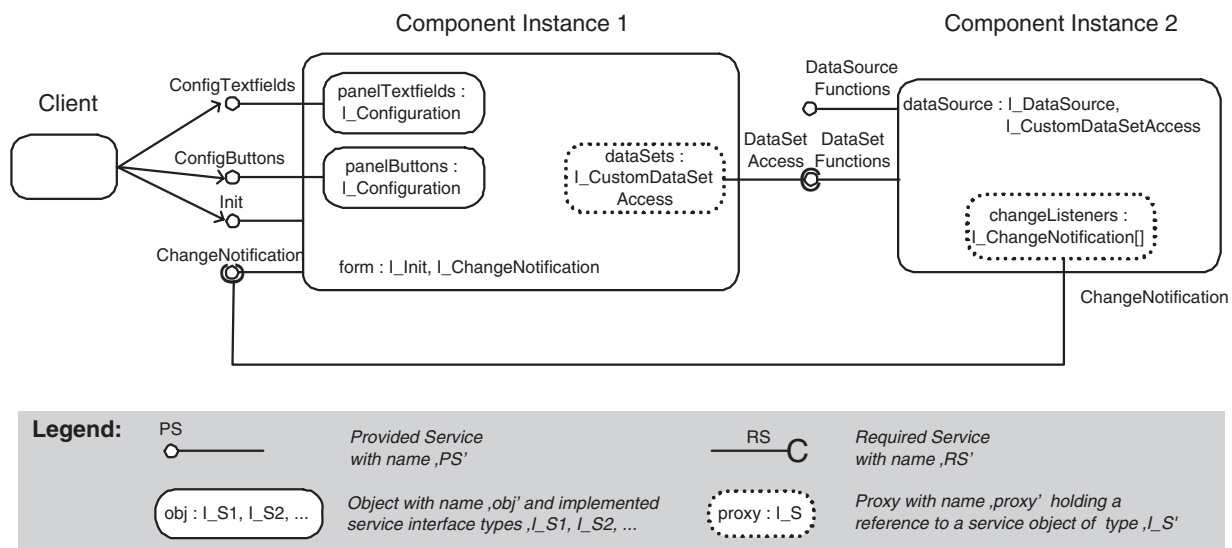


Figure 4.1: Component Instances with their Clients and Implementing Objects

Figure 4.1 shows two interconnected component instances working together to collect customer data. Component instance 1 represents the form used for data input as shown in Figure 4.2. Component instance 2 represents the data source. Component instance 1 stores data inserted into its edit fields to a data source by calling the operations belonging to the service DataSetFunctions of component instance 2. Component instance 2 notifies component instance 1 about changes made to the data source by other users so that component instance 1 can update its edit fields accordingly.

Component instance 1 has four **provided services** ConfigTextfield and Config Buttons of type I_Configuration, Init of type I_Init and ChangeNotification of type I_Change Notification as well as one **mandatory required service** DataSet Access of type I_CustomDataSetAccess.

Figure 4.2: Customer Form to enter Customer Data

The provided services `ConfigTextfields` and `ConfigButtons` enable some kind of configuration of the customer form by clients. These services allow one e.g. to change the text color for all headings of the edit fields or all button labels simultaneously. The provided service `Init` has to be used, if a new data source is selected for storing and retrieving customer data sets. It provides an initialization operation which loads the first data set, initializes the edit fields accordingly and initializes the activation state of the buttons. `ChangeNotification` accepts messages concerning changes made to the data source. Data changes cause the contents of the form's edit fields to be updated.

The **service objects** belonging to the provided services of component instance 1 are the objects named `panelTextfields`, `panelButtons` and `form`. `form` implements two services simultaneously: `Init` and `ChangeNotification`.

The **proxy** for the required service `DataSetAccess` of component instance 1 is named `dataSets`. It holds a reference to the service object for the provided service `DataSetFunctions` of component instance 2 that is, `DataSetAccess` is connected to `DataSetFunctions`. This reference allows component instance 1 to store the customer data inserted in the form-window by calls to the methods belonging to `DataSetFunctions`.

Component instance 2 is a **suitable service provider** with respect to the required service `DataSetAccess` of component instance 1, as it provides the service `DataSetFunctions` of type `I_CustomDataSetAccess` which is equal to the service interface type of `DataSetAccess`.

Component instance 2 has two provided services `DataSourceFunctions` of type `I_DataSource` and `DataSetFunctions` of type `I_CustomDataSetAccess` as well as one **optional required service** `ChangeNotification` of type `I_ChangeNotification`.

The service `DataSourceFunctions` provides operations to change from one data source to another. The service `DataSetFunctions` provides operations to load and store customer data sets from and to a data source. All provided services of component instance 2 are implemented by the service object `dataSource`.

Component instance 2 notifies component instance 1 of changes made to the data source by using its **proxy** `changeListeners`. This proxy stores a reference to the service object for `ChangeNotification` of component instance 1 which accepts messages concerning changes made to the data source. Change notification is e.g. useful, if the data source is accessed by different users simultaneously.

Part of the program code belonging to this description can be found in example 4.1.14 on page 93.

Now the concepts of the component model are described more precisely.

Term 4.1.2 (Service) *A **service** is an access point through which a part of the component's functionality can be accessed by clients or through which the component accesses another component's functionality. The access is restricted to a set of operations defined by the **service interface type**. The service interface type only declares method signatures. It does not carry any implementation. A service also has a **name** which distinguishes it from other services which may have the same service interface type.*

The services of a component are divided into a set of *provided services* and a set of *required services*.

Term 4.1.3 (Provided Service) *A provided service is a service the component implements and provides to its clients.*

Term 4.1.4 (Required Service) *A required service is a service the component does not implement itself but expects to be provided to it by other components.*

A required service may be declared for several reasons already discussed in Section 1.1. One reason is that the declaring component really needs the service to be provided to it by other components to fulfill its task. On the other hand, a required service may be used to notify other components by calls to their provided methods or to declare that the functionality of the component can be extended by the functionality declared by the required service.

The name spaces for provided and required services are separated. Therefore, one component may have a provided and a required service with the same name. This is especially reasonable, if a component only hands over a service of another component. That is, a component *A* provides a service *S* to its clients which is not implemented by the component itself, but is instead provided by another component connected to *A* via a required service. This required service would usually also be named *S*. Such “hand over” behavior is often needed in layered architectures where one layer may only access its direct neighbors.

The required services of a component express its explicit dynamic dependency on other components (see Section 2.1). To resolve these dependencies, a third party (e.g. an assembly tool) is needed which binds the requirements to suitable offerings of other components. Such bindings are done on component instance level and are in the following referred to as connections. Later on we shall explain in more detail, what it means for a component instance to be connected to another one. A component may also have hard wired dependencies on the provided services of other components in its code as far as no other party is needed to create an instance of the needed component and to pass it to the requiring component (see *static dependencies* in Section 2.1). This is

usually the case, if a component consists of several subcomponents. When the composite component is instantiated, instances of the subcomponents are also created which are owned by the composite.

To simplify the way of speaking, we additionally introduce the term of a (*suitable*) *service provider*.

Term 4.1.5 (Suitable Service Provider) *A component which provides services to its clients is called **service provider**. A service provider is called **suitable** with respect to a specified required service R of another component, if it provides at least one service which implements all operations needed by R ¹.*

Term 4.1.6 (Mandatory/Optional Required Service) *A required service R of a component C is called **mandatory**, if R must be connected to an instance of a suitable service provider for every instance of C . Otherwise R is called **optional**.*

E.g., the wordprocessor kernel in Section 3.1 has two mandatory required services: one to a data manager and one to an editor. A connection to a spell checker is optional.

To express that a required service R is optional or mandatory, we introduce a *lower limit* on the number of possible connections, in the following referred to as min_R . If $min_R = 0$, then a connection of this required service to some provided service is optional, otherwise mandatory. Furthermore, $min_R > 0$ allows us to express that a component needs at least min_R service providers to be connected to its required service R to work properly. We also introduce an *upper limit* on the number of connections. For a required service R which allows an unbounded number of service providers to be connected to it, the upper limit max_R can be set to '*'. If lower and upper limit both equal 1, then exactly one connection is allowed and this connection has to be established.

Example 4.1.7 (Wordprocessor Connections) *An instance of the wordprocessor kernel (see example 3.1.1 on page 72) needs a connection to an instance of a data manager as well as to an editor component, but connections to several instances of a data manager or editor are not allowed. That is, the connections to data manager and editor are both limited by $min_R = 1$ and $max_R = 1$. On the other hand, a connection to an instance of the spell checker is optional, but it is possible to connect an arbitrary number of spell checkers to the wordprocessor kernel. Therefore, for this connection $min_R = 0$ and $max_R = *$.*

Term 4.1.8 (Service Object) *A **service object** is an object which belongs to a provided service and which implements the operations declared by the service interface type. For every provided service of a component, a service object is created and bound to the name of the provided service on component instantiation time.*

¹In subsequent chapters *service provider* is also used as a synonym for an *instance of a service provider*. The synonym is used in situations where it is clear that we talk about instances instead of components.

A service object may correspond to several provided services. That is, the names of these provided services are all bound to the same service object. In this case, the service object has to implement all the interface types of the services, it represents.

The possibility to declare different provided services of the same service interface type has several advantages:

- By using different services of the same service interface type it is possible to provide different implementations of the same set of operations. In Figure 4.1 on page 86 `ConfigTextfields` and `ConfigButtons` are examples for different services having the same service interface type. Internally, they refer to different service objects (`panelTextfields` and `panelButtons`) on which the operations are invoked.
- A component can provide different versions of an implementation for the same service interface type. This enables a component to provide an old and a new version of an implementation simultaneously. So clients relying on an old version can use the component as well as clients accessing the new version.
- It might be needed to provide separate access to different instances of the same class contained in an enclosing component instance. If a visual component like a frame contains e.g. two buttons which should be configurable separately from the outside, the visual component has to provide separate configuration services for each button.

Required services are represented by proxies instead of service objects.

Term 4.1.9 (Proxy) A *proxy* is an entity which belongs to a required service of a component and which is capable to hold at least one reference to a service object implementing the requested service interface type. Operations belonging to a required service are called on its proxy. These calls result in calls to the operations of the service object(s) the proxy references.

If a component is instantiated that declares a service as required from another component, it is not yet capable to call operations belonging to this required service since the corresponding proxy does not yet hold a valid reference to a suitable service object. Such a reference must be passed to the proxy prior to a call ².

Therefore, every component instance *A* with a required service *R* has a predefined means to pass a reference to a service object of another component instance *B* (the service provider) on the proxy corresponding to *R*. The process of storing this reference is called *establishing a connection* and is performed by a so-called *connection point object* provided by *A*. If a connection is established, that is, if the proxy holds a reference to

²A proxy can be compared to a private attribute/field of a class to which, at instantiation time, no object reference is assigned. An object reference has to be assigned explicitly by calling a corresponding setter-method.

a suitable service object, this reference can be used for subsequent calls to the implemented operations. Connections always take place on instance level. The connection point object does its job by providing special operations capable of taking a reference to a suitable service object as input which is in turn passed on to the proxy. The use of connection point objects to establish a connection is demonstrated in example 4.1.14. The terms concerning connections used so far are now introduced more precisely.

Term 4.1.10 (Connection) *Let R be a required service of a component instance A and S be a provided service of a component instance B . Then R is called to be **connected** to S , if the proxy belonging to R holds a reference to the service object belonging to S .*

Sometimes we simply say that two component instances are connected to each other without specifying the services involved in this connection. If we want to stress the required service R involved in a connection, but are not yet interested in the actual provided service, we say that R is connected to an instance of B .

Term 4.1.11 (Connection Point Object) *A **connection point object** is an object which belongs to a required service of a component instance and which implements the operations used to establish a connection between the proxy of the required service and a suitable service object or to disconnect a previously established connection.*

The following Figure shows a connection point object CPO implementing the method `connect_R` which is used to store a reference to a suitable service provider sp for the required service R .

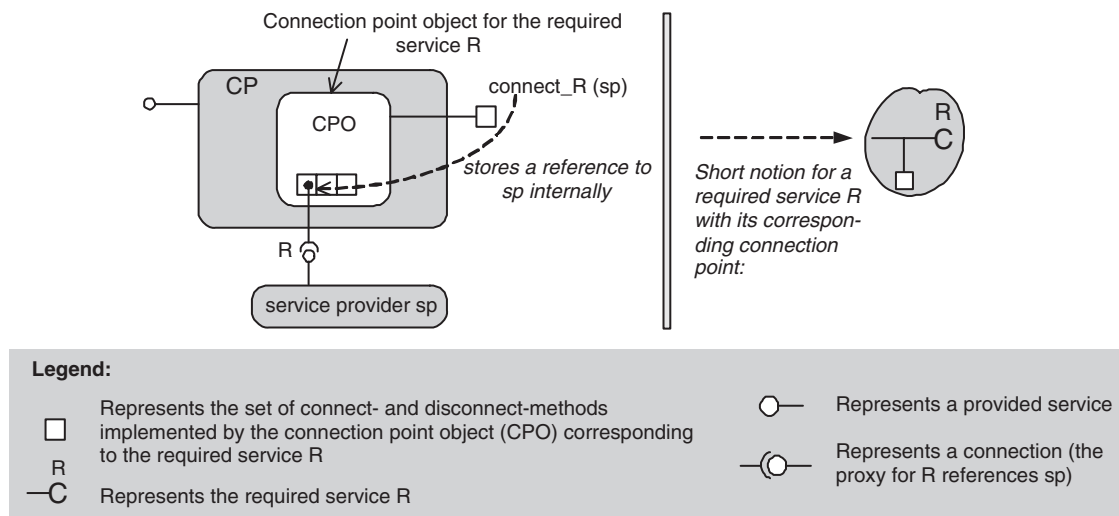


Figure 4.3: Required Services with Connection Points

The stored reference can be used for subsequent calls to sp as long as no disconnect method is called on CPO to disconnect the previously established connection³.

³In contrast to COM, our connection point objects are only forced to implement the specified connect- and disconnect methods of the corresponding required service, not a predefined interface like *IConnectionPoint*, which is the same interface for all types of outgoing interfaces. COM's approach would be too restrictive to be able to integrate arbitrary existing Components Off The Shelf.

Disconnect-methods are needed for dynamic topologies or at assembly time to reconfigure an existing assembly, if the assembly is supported by a tool.

Every connect-method must have at least one parameter having as type the service interface type of the required service. When calling such a method, a reference to a service object implementing the required methods must be passed as the corresponding actual parameter (see example 4.1.14). Other parameters may exist additionally. To every connect-method there exists a corresponding disconnect-method to disconnect the previously established connection. There are no constraints on the names or the number of parameters of the connect- and disconnect-methods. Due to technical reasons explained later on in the thesis, only the following constraints exist: for every connect-method there must be exactly one parameter of the service interface type of the required service and all other parameters have to be of primitive data types. For disconnect-methods a parameter having as type the service interface type is not mandatory.

Term 4.1.12 (Connection Point Interface) *The connect- and disconnect-methods implemented by the connection point object are grouped by a **connection point interface**.*

In example 4.1.14 on page 93, form, an instance of `CustomerForm`, is the connection point object for the required service `DataSetAccess` of component instance 1. It implements the connection point interface `I_ConnectionPoint_CustomDataSetAccess`, the interface declaring the operations used to establish or disconnect a connection. An example for a valid connect-methods is

```
public void connectDataSource(I.CustomDataSetAccess dataSource);
```

from example 4.1.14 on page 93. A call to a connect-method is similar to the registering of a listener at an event source in the JavaBeans component model (see Section 2.2.1.1).

Besides the services providing part of the functionality of a component, every component has to implement a special *service access interface* to provide a standardized way to get a reference to

1. the service object implementing the methods belonging to a provided service and
2. the connection point object belonging to a required service. (A reference to the connection point object is needed to connect the required service to an instance of a suitable service provider.)

Term 4.1.13 (Service 'ServiceAccess') *The service named **ServiceAccess** is a provided service with corresponding service interface type **I.ServiceAccess**. `I.ServiceAccess` declares two methods: `getServiceReference` and `getConnectionPointObject`. `getServiceReference` takes as its parameter the name of a provided service and returns a reference to the service object bound to the provided service. `getConnectionPointObject` takes as its parameter the name of a required service and returns a reference to the connection point object which belongs to the required service.*

In Java the type of the service access interface may be declared as follows:

```

interface I_ServiceAccess {
    // returns a reference to the service object
    // implementing the provided service requested
    // by a client
    Object getServiceReference (String PServiceName);

    // returns a reference to the connection point
    // object of the requested required service
    Object getConnectionPointObject (String RServiceName);
}

```

The following code example demonstrates the implementation of components, the access to provided services and the process of connecting a required service of a component instance to a suitable provided service of another component instance. The example shows Java code corresponding to example 4.1.1. For simplicity, the example is entirely written in Java. Instantiation of components is simulated using Java's *new*-operator. In general, instantiation will include additional activities like loading of persistent data as in the JavaBeans component model for example (see Section 2.2.1.1, item 4 on page 21).

Example 4.1.14 (Creation of Component Instances, Service Access and Interconnections)

```

/*-----*/
/* Service interface type for 'ServiceAccess' */
/*-----*/

interface I_ServiceAccess {
    Object getServiceReference (String PServiceName);
    Object getConnectionPointObject (String RServiceName);
}

/*-----*/
/* Service interface types for provided and required services */
/*-----*/

public interface I_Configuration {
    public void setTextColor (java.awt.Color c);
    public java.awt.Color getTextColor ();
}

public interface I_Init {
    public void initialization ();
}

public interface I_ChangeNotification {
    public void dataSetChanged(CustomDataSet dataSet);
}

public interface I_DataSource {
    public void openDataSource (String name);
    public void closeDataSource (String name);
    public void createDataSource (String name);
}

public interface I_CustomDataSetAccess {
    public CustomDataSet getFirst();
}

```

```

    public CustomDataSet getNext();
    public CustomDataSet getByID(int customerID);
    public void setByID(CustomDataSet dataSet);
    // ...
}

/*-----*/
/* Types of used connection points */
/*-----*/

public interface IConnectionPoint_ChangeNotification {
    public void addNotificationListener(I_ChangeNotification listener);
    public void removeNotificationListener(I_ChangeNotification listener);
}

public interface IConnectionPoint_CustomDataSetAccess {
    public void connectDataSource(I_CustomDataSetAccess dataSource);
    public void disconnectDataSource();
}

/*-----*/
/*           Helper classes           */
/*-----*/

public class CustomDataSet {
    public int customerID;
    public String name;
    public String surname;
    public String phoneNumber;
    // ....

    public CustomDataSet () {
        // Initialize with default data.
    }
    public CustomDataSet (int customerID, String name, String surname, String phoneNumber) {
        // ...
    }
}

public class NamedTextField extends java.awt.Panel {
    java.awt.Label label = new java.awt.Label();
    java.awt.TextField textfield = new java.awt.TextField (12);

    public NamedTextField (String head) {
        // Add 'label' and 'textfield' to 'this' and initialize 'label' with 'head'.
    }
}

/*-----*/
/* Implementations of service objects other than the component instances */
/*-----*/

public class PanelButtons extends java.awt.Panel implements I_Configuration {
    private java.awt.Color textColor = java.awt.Color.BLACK;

    // Declare buttons btNext, btPrevious, btNew

    public PanelButtons () {
        this.setLayout(new java.awt.FlowLayout());
        // Add buttons to 'this'
    }
}

```

```

    /*** Implementation of I_Configuration ***/
    public void setTextColor (java.awt.Color c) {
        textColor = c;
        // set the foreground color of all buttons to c
    }
    public java.awt.Color getTextColor () {
        return textColor;
    }
}

public class PanelTextFields extends java.awt.Panel implements I_Configuration {
    private java.awt.Color textColor = java.awt.Color.BLACK;

    // Declare tfCustomerID, tfName, tfSurname, tfPhoneNumber of type NamedTextField

    public PanelTextFields () {
        this.setLayout(new java.awt.FlowLayout());
        // Add the declared textfields to 'this'
    }

    public CustomDataSet getCustomDataSetFromForm() {
        // ...
        return new CustomDataSet(tfCustomerID.textfield.getText(), tfName.textfield.getText(),
                                tfSurname.textfield.getText(), tfPhoneNumber.textfield.getText());
    }

    public void setCustomDataSetToForm(CustomDataSet dataSet) {
        // Display the entries of 'dataSet' in the corresponding textfields
    }

    // ...

    /*** Implementation of I_Configuration ***/
    public void setTextColor (java.awt.Color c) {
        textColor = c;
        // set the foreground color of all textfield labels to c
    }
    public java.awt.Color getTextColor () {
        return textColor;
    }
}

/*-----*/
/* Implementation of component 1 */
/*-----*/

import java.awt.*;
import java.awt.event.*;

public class CustomerForm extends Panel implements
    I_ServiceAccess, I_Init, I_ChangeNotification,
    I_ConnectionPoint_CustomDataSetAccess {
    // The provided services 'Init' and 'ChangeNotification'
    // have the same service object namely 'this'.

    // The service objects for the provided services 'ConfigTextfields' and
    // 'ConfigButtons' are:
    private PanelTextFields panelTextfields = new PanelTextFields();
    private PanelButtons panelButtons = new PanelButtons();

    // Proxy for the required service 'DataSetAccess'.
    // The corresponding connection point object is 'this'.
    private I_CustomDataSetAccess dataSets = null;

```

```

public CustomerForm () {
    // Set background color and layout manager.
    // Add textfields and buttons to the form.
    // Initialize the textfields and the button activation state.

    // Register 'ActionListeners' to buttons
    panelButtons.btNext.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            // Get actual dataSet from form and store it to data base or file.
            // Read next dataSet from data base or file and assign its
            // attributes to the edit fields of the form ...
            // Change the activation state of the buttons.
        }
    });

    // ....
}

/**** Implementation of I_ServiceAccess ****/
public Object getServiceReference (String PServiceName) {
    if (PServiceName.equals("Init") || PServiceName.equals("ChangeNotification"))
        return this;
    if (PServiceName.equals("ConfigTextfields")) return panelTextfields;
    if (PServiceName.equals("ConfigButtons")) return panelButtons;
    return null;
}
public Object getConnectionPointObject (String RServiceName) {
    if (RServiceName.equals("DataSetAccess")) return this;
    return null;
}

/**** Implementation of I_Init ****/
public void initialization () {
    // Read first dataset from data source (data base or file)
    // Assign the dataset to the edit fields of the form
    // Disable Previous-Button ....
}

/**** Implementation of I_ChangeNotification ****/
public void dataSetChanged(CustomDataSet dataSet){
    // Assign the dataSet to the edit fields of the form, if customerID from
    // 'dataSet' is equal to the customerID shown in the corresponding edit field.
    // ...
}

/**** Implementation of IConnectionPoint_CustomDataSetAccess ****/
public void connectDataSource(I_CustomDataSetAccess dataSets) {
    this.dataSets = dataSets;
}
public void disconnectDataSource(){
    this.dataSets = null;
}

}

/*-----*/
/* Implementation of component 2 */
/*-----*/

public class CustomDataSource implements I_ServiceAccess, I_DataSource, I_CustomDataSetAccess,
                                         IConnectionPoint_ChangeNotification {
    // Here 'this' is the service object for the provided services
    // 'DataSourceFunctions' and 'DataSetFunctions'.

```

```

private int maxListeners = 10;
private int registeredListeners = 0;

// Proxy for the required service 'ChangeNotification' with service interface type
// 'I_ChangeNotification'. The corresponding connection point object is 'this'.
private I_ChangeNotification[] changeListeners = new I_ChangeNotification[maxListeners];

private int registered (I_ChangeNotification listener) {
    // Check whether 'listener' is already in 'changeListeners'.
    // If true, return the corresponding array index otherwise return -1
    // ...
}

/**** Implementation of I_ServiceAccess ****/
public Object getServiceReference (String PServiceName) {
    if (PServiceName.equals("DataSourceFunctions") ||
        PServiceName.equals("DataSetFunctions")) return this;
    return null;
}
public Object getConnectionPointObject (String RServiceName) {
    if (RServiceName.equals("ChangeNotification")) return this;
    return null;
}

/**** Implementation of I_DataSource ****/
public void openDataSource (String name) {
    // Open data base or file.
}
public void closeDataSource (String name) {
    // Close data base or file.
}
public void createDataSource (String name) {
    // Create new data base or file.
}

/**** Implementation of I_CustomDataSetAccess ****/
public CustomDataSet getFirst() {
    // Load first data set from data base or file.
}
public CustomDataSet getNext() {
    // Load next data set from data base or file.
}
public CustomDataSet getByID(int customerID) {
    // Load data set identified by 'customerID' from data base or file.
}
public void setByID(CustomDataSet dataSet) {
    // Store 'dataSet' to data base or file.
    // Notify change listeners:
    for (int i = 0; i < registeredListeners; i++) {
        changeListeners[i].dataSetChanged(dataSet);
    }
}

/**** Implementation of IConnectionPoint_ChangeNotification ****/
public void addNotificationListener(I_ChangeNotification listener) {
    if (registeredListeners < maxListeners && registered(listener) == -1) {
        changeListeners[registeredListeners] = listener;
        registeredListeners++;
    }
}
public void removeNotificationListener(I_ChangeNotification listener) {
    // ...
}
}

```



```

/*-----*/
/* Generation and connections of component instances */
/* and service access */
/*-----*/

import java.awt.*;
import java.awt.event.*;

public class InterconnectedComponentInstances {

    public static void main (String[] argv) {
        // Generation of component instances.
        I_ServiceAccess form = new CustomerForm();
        I_ServiceAccess dataSource = new CustomDataSource();

        // Connect the required service 'DataSetAccess' of component instance 1 (form) to
        // the provided service 'DataSetFunctions' of component instance 2 (dataSource).
        IConnectionPoint_CustomDataSetAccess cp_DataSetAccess =
            (IConnectionPoint_CustomDataSetAccess)form.getConnectionPointObject("DataSetAccess");
        I_CustomDataSetAccess dataSets =
            (I_CustomDataSetAccess)dataSource.getServiceReference("DataSetFunctions");
        cp_DataSetAccess.connectDataSource(dataSets);

        // Connect the required service 'ChangeNotification' of component instance 2 to
        // the provided service 'ChangeNotification' of component instance 1.
        IConnectionPoint_ChangeNotification cp_ChangeNotification =
            (IConnectionPoint_ChangeNotification)dataSource.getConnectionPointObject("ChangeNotification");
        I_ChangeNotification cn = (I_ChangeNotification)form.getServiceReference("ChangeNotification");
        cp_ChangeNotification.addNotificationListener(cn);

        // Get access to the provided service 'Init' of 'form' and call initialization()
        // to load the first dataSet etc.
        I_Init initObject = (I_Init)form.getServiceReference("Init");
        initObject.initialization();
        //....

        // Get access to the provided services 'ConfigTextfields' and 'ConfigButtons' of 'form' and call the
        // of 'form' and call the corresponding methods belonging to these services.
        I_Configuration conf = (I_Configuration)form.getServiceReference("ConfigTextfields");
        conf.setTextColor(Color.BLACK);
        conf = (I_Configuration)form.getServiceReference("ConfigButtons");
        conf.setTextColor(Color.BLUE);
        //....

        // The following code is used to visualize the form by adding it to a frame
        Frame f = new Frame();
        f.setSize (500,200);
        f.setLocation (100,100);
        f.setLayout(new FlowLayout());

        f.add((CustomerForm)form);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosing (WindowEvent e) {
                // Programm beenden
                System.exit (0);
            }
        });
        f.setVisible (true);
    }
}

```

4.1.1.2 Plug, A Higher Level Concept

Up to now, we only considered access and interconnections on service level. These kinds of interconnections are still quite fine-grained and can not reflect the following situations:

- Two components need mutual access to their provided services, so a bi-directional connection is needed. (See e.g. Figure 4.1, services `DataSetFunctions` and `Change Notification`.)
- Several services of a component are semantically related and have to be used by clients as a whole.

For this purpose we introduce the notion of a plug:

Term 4.1.15 (Plug) *A **Plug** is a named unit of interconnection consisting of a set of semantically related required and/or provided services of the same component defining a certain communication protocol. All plugs of the same component have to represent disjoint sets of services and are identified by their names.*

Being a unit for interconnections means the following:

1. A client of a provided service belonging to a plug *Pl* of a component instance *C* is also a client of all other provided services belonging to *Pl*.
2. The client **is forced** to be **itself** the provider for all required services of *C* which belong to *Pl*. Thus, the client has to agree on the same communication protocol as defined by *Pl*.

A plug allows one to define interconnections and subtyping on a higher level of abstraction than on service level. To the best of our knowledge, this novel feature does not exist in any of the existing industrial component models nor in research models. It is demonstrated in the examples 4.1.23 and 4.2.1.

In the following, some examples are given motivating the use of plugs. The first example comes from an existing application in the area of packaging machines. The second one is based on a well-known design pattern: the MVC (Model View Controller) pattern. The third example deals with compound documents. Services are denoted by `ServiceName: I_ServiceType`. For corresponding provided and required services the same names and service interface types are assumed, although weaker conditions are possible (see Section 4.4.1). In the figures this is denoted by only one occurrence of `ServiceName: I_ServiceType` at the point where the two services are connected. For simplicity, the service `ServiceAccess` which has to be provided by every component is omitted. Services which would be good candidates to be grouped by a plug are visually surrounded by a dotted ellipse.

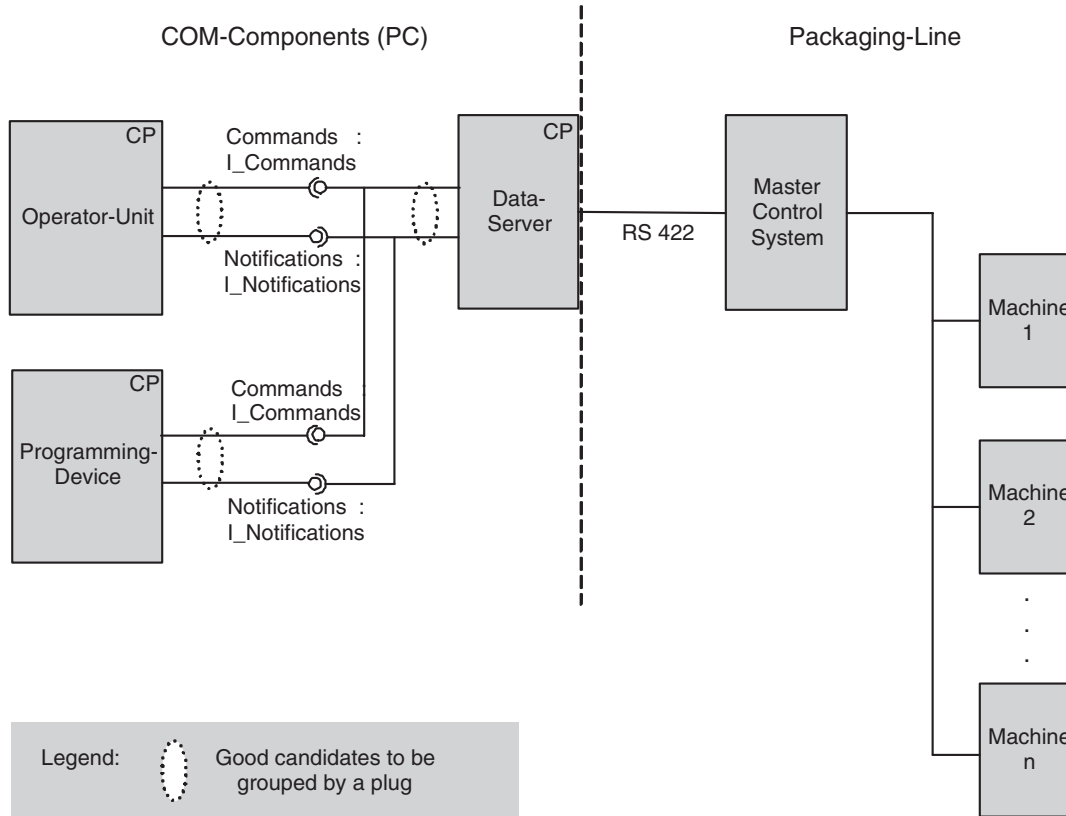


Figure 4.4: Packaging Machines

Example 4.1.16 (Packaging Machines) A set of packaging machines working together to pack the same product, is called a **line**. A line is controlled by a master control system which is connected to all machines in the line. It serves to synchronize the work of all these machines. It gives commands to each machine, reacts on errors by special shut down mechanisms, it retrieves data from the connected machines etc. It also interacts directly with the data-server component (see below), one of the components provided for human users to interact with the system. A human operator changes the operation mode of a machine directly by pressing a button on a panel directly connected to the machine. The master control system is notified of such changes which must in turn be transmitted to the data-server.

On the user side, there are three main components implemented by COM-components: an operator-unit, a programming-device and a data-server. The operator-unit visualizes selected states and signals of the connected packaging machines and provides functionality to the operator for communication with the master control system. The operator can start production (that is, the process of packaging a special product is started), he can give halt commands (that is, the process of packaging products is halted for a while), he can start loading new programs to the connected machines which is necessary, if a new product will be packaged, he can explicitly ask

for the state of signals (e.g. vacuum) of a selected machine etc. The operator-unit uses the data-server as the interface to the master control system. The data-server is used to send commands and data over a RS422-interface to the control system and to retrieve interesting data, as e.g. the actual state of each machine. It polls the connected master control system and notifies the operator-unit and possibly connected programming devices of data they are interested in. The programming-device component is used to write new packaging programs, to compile them, and to send them to the control systems. It may also be used to retrieve data from the connected machines, to signal errors etc. The programming-device component also uses the data-server component for the communication with the master control system.

The operator-unit and the data-server need mutual access to provided services of each other (Commands and Notifications). The same holds for the programming device and the data-server. In this case, four connections have to be established: one between the required service Commands of the operator-unit and the provide service Commands of the data-server, the second one between the provided service Notifications of the operator-unit and the required service Notifications of the data-server, the third one between the required service Commands of the programming-device and the provide service Commands of the data-server, the fourth one between the provided service Notifications of the programming-device and the required service Notifications of the data-server. Using plugs, the number of connections can be reduced to two plug-connections. The first plug groups the required service Commands and the provided service Notifications of the operator-unit resp. the programming-device. The second plug groups the provided service Commands and the required service Notifications of the data-server⁴. Besides reducing the number of connections, the use of plugs ensures that the data-server would in any case be reconnected to the **appropriate client**. This is essential since each client can announce its individual data of interest. Thus a client only wants to be notified, if changes occur to its own data of interest. Without using plugs, it can even happen that the data server is not reconnected to the client; thus the client is not notified at all.

We shortly sketch the interface types of the services Commands and Notifications to get an idea of what is going on between the data-server and its clients. The service interface types are denoted as Java types.

```
interface I_Commands {
    void startPolling (/*...*/);
    void stopPolling (/*...*/);
    byte[] getSignals (String machine);
    void setSignal (String machine, int signalNumber, byte value);
    int getOperationMode (String machine);
    void setOperationMode (String machine, int operationMode);
    void transferLineData (byte[] lineData);
    void transferProgramData (String machine, byte[] program);
    void selectMachinesOfInterest (String[] machines, int clientID);
    /* .... */
}

interface I_Notifications {
    void operationModeChanged (String machine, int operationMode);
```

⁴See the dotted ellipses in Figure 4.4.

```

void emergencyStopPressed ();
void newSignals (String machine, byte[] signals);
/* .... */
}

```

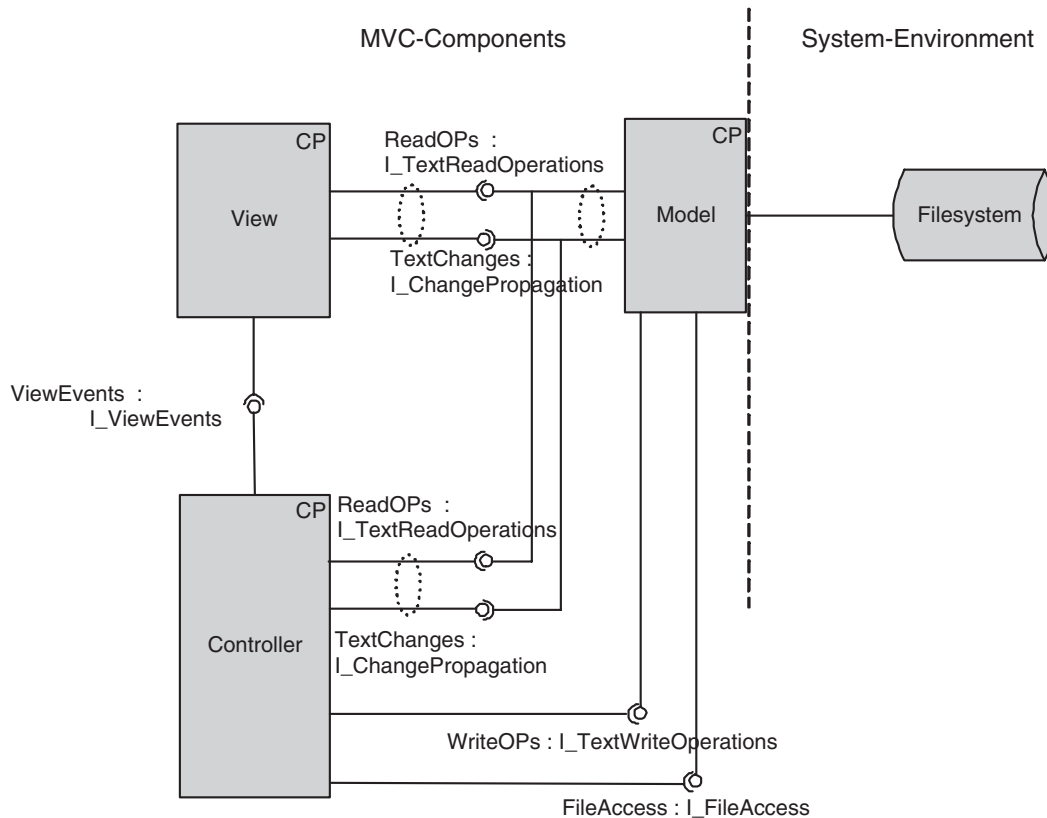


Figure 4.5: Model View Controller

Example 4.1.17 (Model View Controller) This example describes a model view controller architecture, the classical pattern underlying graphical user interfaces. It distinguishes three kinds of components:

- The **model** which represents the application data. In our example, the model represents text to be edited by a user. The model provides functionality to read and write text, to load text from a file, to store text to a file etc. For this purpose, it provides the services `ReadOps`, `WriteOps` and `FileAccess`. It notifies views and controllers of changes on the state of the model. Therefore, it uses the required service `TextChanges`.
- The **views** display the model or part of it on the screen. Different views can show different perspectives of the model. In our example a view provides functionality by the provided

service TextChanges which allows the view to be notified by the model, if text changes (provided service TextChanges) occur. If notifications about text changes occur, the view in turn reads data from the model for necessary updates. For this purpose, it uses the required service ReadOPs. A view also has a required service ViewEvents to notify a connected controller about user actions.

- *The **controllers** control the interaction between users and the model. If a controller is notified by a view of user actions triggering operations on the model, the controller calls the corresponding methods on the model. For this purpose, it uses its required services ReadOPs, WriteOPs and FileAccess. Its provided service TextChanges is used by the model to notify the controller of state changes of the model which in turn reads the modified state. The notification of state changes of the model is especially needed, if several controllers exist.*

Similar to the first example, the model and a view as well as the model and a controller need mutual access to the provided services ReadOPs and TextChanges of each other. The number of needed connections depends on the number of views and controllers in the system. Using plugs grouping the services ReadOPs and TextChanges the number of connections to be established could significantly be reduced. As in the first example, the use of plugs ensures that the model is reconnected in each case to the appropriate client (view, controller) which is essential for the clients to work properly. Without being notified of text changes, a view could e.g. not display the actual state of the model.

The interface types of the services denoted in Figure 4.5 are shown below.

```
interface I_FileAccess {
    boolean loadFile (String fileName);
    boolean storeFile (String fileName);
    void newFile ();
}

interface I_TextWriteOperations {
    void writeChar (int pos, char ch);
    void insertChar (int pos, char ch);
    void deleteChar (int pos);
    void writeString (int pos, String st);
    /* ....*/
}

interface I_TextReadOperations {
    char readChar (int pos);
    String readString (int pos, int length);
    String readWholeText ();
    /* ....*/
}

interface I_ChangePropagation {
    void newText ();
    void charInserted (int pos, char ch);
}
```

```

void charDeleted (int pos);
void stringInserted (int pos, String st);
/* ....*/
}

interface I_ViewEvents {
    void charTyped (int pos, char ch);
    void charDeleted (int pos);
    void newFileRequested();
    void loadFileRequested(String fileName);
    /* ....*/
}

```

Example 4.1.18 (Compound Documents) *Compound documents⁵ enable users working within a single application to manipulate data written in various formats and derived from multiple sources. For example, a user might insert into a word processing document a graph created in a second application and a sound object created in a third application. Activating the graph causes the second application to load its user interface, or at least that part containing tools necessary to edit the object. Activating the sound object causes the third application to play it. In both cases, a user is able to manipulate data from external sources from within the context of a single document.*

One distinguishes document containers and document servers. Document servers maintain data and provide means to display and manipulate these data. Examples for document servers are e.g. the graph application and the sound object application from above. Document containers do not have their own data. But they can integrate different parts provided by document servers. Every part gets its own display area. Document containers can also be document servers. In this case they also come with their own data. An example is the word processing application from above that is able to integrate graphs and sound objects into a word processing document.

As everything can be edited where it is displayed, the user has the illusion to edit a single document. Such behavior requires a high cooperation of the applications involved. To reach this goal, the cooperation can be realized as follows. Every server provides a content object for every part of a container representing data of this server. On the other hand, a container provides a client site object for every content object of one of its parts. Both objects interact as depicted in Figure 4.6.

Some of the methods belonging to the service interface types involved in the interaction between both objects are described below. Instead of showing exact method signatures, the method name is denoted as well as an explanation of its functionality. Often used call sequences could be:

```

I_Object.Initialize → I_ClientSite.ShowObject
I_Object.Close → I_ClientSite.SaveObject,
                I_Advise.OnClose

```

⁵Some of the following text can be found in Microsoft's MSDN Library.

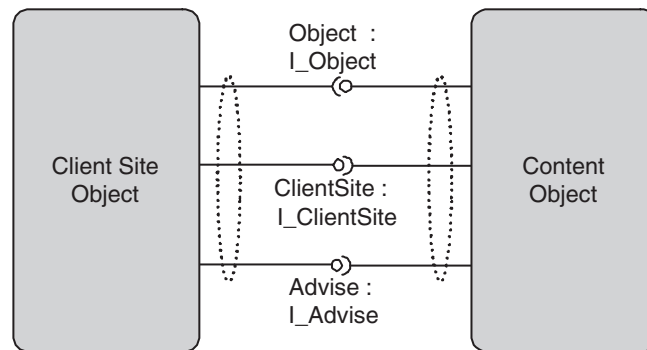


Figure 4.6: Cooperating Objects in Compound Documents

Methods of I.ClientSite

SaveObject
ShowObject
OnShowWindow
NewObjectLayout

Comment

Saves embedded object.
Asks container to display object.
Notifies container when object becomes visible or invisible.
Asks container to resize display site.

Methods of I.Advise

OnDataChange
OnViewChange
OnRename
OnClose

Comment

Advises that data has changed.
Advises that view of object has changed.
Advises that name of object has changed.
Advises that object has been closed.

Methods of I.Object

Close

Initialize
DoAction

Comment

Moves object from running to loaded state.
Following such a call, the object still appears in its container, but is not open for editing.
Initializes embedded object from selected data.
Invokes object to perform one of its enumerated actions as e.g. the action that occurs when an end user double-clicks the object in its container.
Sets extent of object's display area. A container calls SetExtent, when it needs to dictate the size at which it will be displayed to an embedded object.
Often, this call occurs in response to an end user resizing the object window.
Retrieves extent of object's display area.
Recommends color scheme to object application. The container recommends a color palette to the object application, which is not obliged to use it.

SetExtent

GetExtent
SetColorScheme

These three closely related services should be grouped by a plug⁶ on both sides. One reason is that the three services are semantically related. The other reason is that a plug is a unit of interconnection. This guarantees that the three connections between a content- and its client site-object are all established and thus guarantee that all calls belonging to a call sequence can be executed.

4.1.1.3 Constraints on Connections

While plugs can be described as a special relation between several services, another relation exists which can be defined between required services only. This relation can be regarded as a constraint concerning proper connections. Such constraints are explicitly stated in the component interface specification described in Section 4.1.2. We considered only one kind of constraint exemplarily. Although a lot of other constraints could be important, it would go beyond the scope of this thesis to provide a general constraint language capable of expressing arbitrary constraints.

Assume that a component instance requires several services of the same type. If there is a component instance providing a service which implements all operations declared by this type, all these required services could be connected to this provided service. But this may result in a malfunctioning or at least in an unexpected behavior. To demonstrate this problem, let us have a look at the following examples.

Example 4.1.19 (Flexible Customer Form) *The component realizing the customer form from Figure 4.2 on page 87 can be made more flexible by enabling a different look and feel for e.g. the buttons contained in the form. Two different kinds of look and feel are shown in Figure 4.7.*

Figure 4.7: Two other Views of the Customer Form

To enable this flexibility, the panel component containing the buttons does no longer create the buttons. Instead, it defines a required service with the service interface type `I_Button` for

⁶See the dotted ellipses in Figure 4.6.

every button. Its corresponding proxies are capable of storing a reference to an instance of a button implementing `I_Button`. Before delivering a customer form component to a customer, buttons having a look and feel desired by the customer have to be selected for instantiation. The three button instances needed have to be configured to match their role by setting appropriate texts and/or icons. Then the required services of an instance of the panel component have to be connected to the appropriate button instances. The resulting configuration is stored and delivered to the customer.

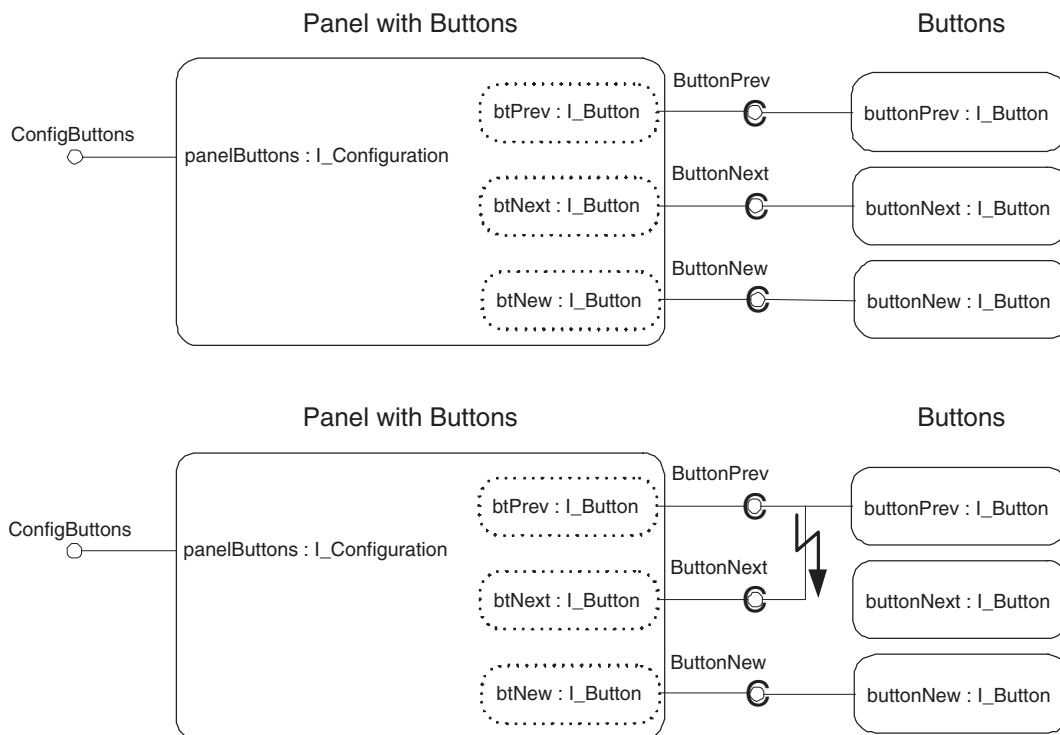


Figure 4.8: Example for Required Different Service Providers

If two or more required services of an instance of the panel component are by a mistake connected to the **same** instance of a button component, the customer form using the panel component can not work properly. Depending on the GUI component used as container for the buttons (panel component), the wrong connection may result in the display of only one or two buttons. Actions corresponding to a mouse click on a button may not be executed properly etc.

Therefore, we should have a means to express for a set of required service of the same component instance that each of these services has to be connected to a service provider which is distinct from all service providers connected to the other required services of the set. This constraint acts as a kind of *alias control*. It declares where aliasing is not allowed.

Term 4.1.20 (Constraint Different Service Providers) *A constraint of kind "Different Service Providers" is represented by a named set of required services of the same component with the following semantics: Every service provider connected to one of the required services belonging to such a **constraint set** must be distinct from every service provider connected to one of the other required services belonging to this set. In addition, constraint sets have to obey the constraint rules 1 and 2 from below.*

All constraints of kind *Different Service Providers* have to obey the following rules:

Constraint Rule 1: Let C be a constraint with constraint set C_{set} and let $M \subseteq C_{set}$ be a set of required services belonging to C_{set} . If R is another required service different from all required services in M ($\{R\} \cap M = \emptyset$) and a service provider for R has to be distinct from all service providers for the required services in M , then R has to belong to C_{set} , too ($R \in C_{set}$).

Constraint Rule 2: Required services belonging to a constraint of kind *Different Service Providers* must not belong to a plug of the same component.

Constraint Rule 1 guarantees that a constraint is uniquely identified by its constraint set only and that it can not be expressed by other constraints with smaller constraint sets holding simultaneously for the same component. Thus, in the remainder of this thesis, we do not distinguish between constraints and constraint sets. Instead of giving a formal proof, we present a short example which intuitively shows the proof idea.

Let C be a constraint with constraint set $C_{set} = \{R_1, R_2, R_3\}$. Then three different service providers have to be selected, one for each required service. The same constraint would be implied by three 'smaller' constraints C_1 , C_2 , and C_3 with their corresponding constraint sets $C_{set}^1 = \{R_1, R_2\}$, $C_{set}^2 = \{R_2, R_3\}$ and $C_{set}^3 = \{R_1, R_3\}$ which hold simultaneously. But C_{set}^i , $1 \leq i \leq 3$, are no valid constraint sets, as each of them breaks the first rule. E.g. R_3 does not belong to C_{set}^1 although a service provider to be connected to R_3 has to be distinct from the service provider connected to R_1 (see constraint set C_{set}^3) as well as from the service provider connected to R_2 (see constraint set C_{set}^2). Thus C is only represented by C_{set} .

Constraint Rule 2 is due to the fact that required services belonging to a plug all have to be connected to services of the *same* component instance, whereas required services belonging to a constraint must be connected to services of *different* component instances.

One might argue that one should generally demand that required services of one component having the same type have to be connected to different service providers. But this would be too restrictive. If there is e.g. a component consisting of several sub-components and, at runtime, every instance of a subcomponent needs a connection to the same component instance giving access to a common data source, this approach would not be adequate. We can use a modified version of example 4.1.16 shown in Figure 4.9 to demonstrate this problem. Here every subcomponent "exports" its required service *I.Commands* from its enclosing component to enable a connection to the external component *Data – Server*.

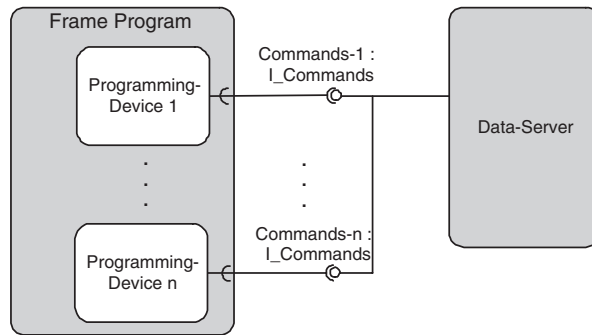


Figure 4.9: Component Instances with several Required Services of the same Type, which have to be connected to a common Service Provider

Here all of the instances *ProgrammingDevice* 1 up to *ProgrammingDevice* *n* have to be connected to the same instance of the *Data-Server* which is the sole entity connected to the master control system controlling all connected machines. The *Data-Server* itself has to be reentrant to serve different programming devices in different states.

4.1.1.4 Summary of Concepts

Most of the concepts described in the previous sections are summarized in Figure 4.10.

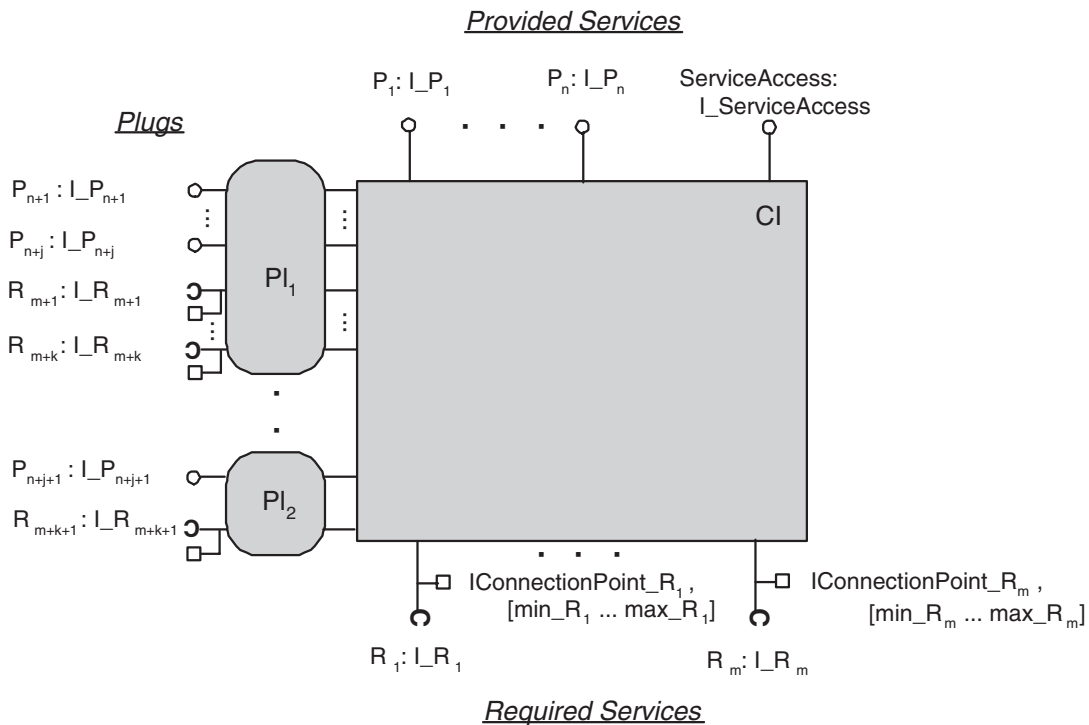


Figure 4.10: Component Model

$P_i : I_P_i$ ($1 \leq i \leq n+j+1$) denotes a provided service with name P_i and corresponding service interface type I_P_i and $R_i : I_R_i$ ($1 \leq i \leq m+k+1$) a required service with name R_i and corresponding service interface type I_R_i . $IConnectionPoint_R_i$ denotes the set of connect - and disconnect - methods provided by the connection point object for the required service named R_i . $[min_R_i...max_R_i]$ denotes the lower limit resp. upper limit for connections to the required service R_i .

Pl_1 denotes a plug grouping the provided services P_{n+1}, \dots, P_{n+j} as well as the required services R_{m+1}, \dots, R_{m+k} thereby defining a semantic relationship between these services. Pl_2 denotes a plug grouping the provided service P_{n+j+1} and the required service R_{m+k+1} . For simplicity, the connection point methods as well as the lower and upper limits were omitted for $R_{m+1}, \dots, R_{m+k+1}$.

Although a component is more than a mere grouping of its stand-alone services and plugs, it manifests itself only by these entities of its component interface.

Term 4.1.21 (Component Interface) A *component interface* consists of the service "ServiceAccess", a further set of stand-alone provided and required services, a set of plugs and a set of constraints of kind "Different Service Providers".

Term 4.1.22 (Component) A *component* is some piece of code which conforms to a given component interface, i.e.:

- The component implements the service access interface.
- For every provided service of the component interface the component returns the corresponding service object when queried for it on its service access interface.
- For every required service of the component interface the component returns the corresponding connection point object when queried for it on its service access interface.

4.1.2 Component Interface Specifications

4.1.2.1 General Specifications

The specification of a component's capabilities and requirements is made available in the form of metadata written in a simple language. Using metadata, such specifications can also be provided for components conforming to other component models which do not support all of our features (e.g. JavaBeans, CCM). The metadata describe a component's interface to the outer world and are therefore called *component interface specification*. Using this specification, the interface of a component can be described uniformly for all known industrial component models as well as for other kinds of components like remote objects or composite UCM-components as introduced in Section 4.2.2. The component interface specification is divided into three main parts.

The first part contains information concerning the naming conventions used to describe the service interface types of the services. These naming conventions depend on the component model used to implement the atomic component which has the specified component interface. For the JavaBeans component model, the service interface types are denoted as Java types, for COM components, they are denoted by IIDs, for .NET

components, they are denoted as .NET types etc. The entries for this part are described in more detail in Section 4.10 on page 182.

The second part contains a description of the component's services (provided and required). The description of a required service includes information on the corresponding connection point methods and on the lower and upper limit for the number of connections.

The third part describes relations on the defined services, as there are for example higher order entities like plugs or the definition of a constraint like *Different Service Providers*. Whereas plugs are intended for groups of related services which are all together used and/or provided by one "partner" component agreeing on the same communication protocol represented by the plug, single services not belonging to a plug can be used or connected independently of other services of the same component.

The specification of a component interface looks as follows:

```

ComponentInterface CI {
  GeneralDescriptions
    NamingConventions = nc      (where nc ∈ {JavaType, .NETType, GUID...}
                                defines the naming conventions for the service
                                interface types of the component's services)

  ServiceDefinitions
    ProvidedServices
      ServiceAccess      :   I_ServiceAccess
      P1                  :   I_P1
      ...
      Pn+j                :   I_Pn+j
      ...
    RequiredServices
      R1                  :   (I_R1, IConnectionPoint_R1, [min_R1...max_R1])
      ...
      Rm+k                :   (I_Rm+k, IConnectionPoint_Rm+k, [min_Rm+k...max_Rm+k])
      ...

  ServiceRelations
    Plugs
      Pl1 =   ({Pn+1, ..., Pn+j}, {Rm+1, ..., Rm+k})
      ...
      Plo =   { ... }

    Constraints
      C1 =   {Ri1, ..., Ril}      (where Rij, 1 ≤ j ≤ l and Rjs, 1 ≤ s ≤ p belong to the set of
      ...                               required services of CI without those belonging to a plug.)
      Cr =   {Rj1, ..., Rjp}
}

```

Figure 4.11: Component Interface Specification

In the component interface specification described above the used notations mean the following:

- CI denotes the name of the component interface.
- P_i denote the names of the provided services, I_{P_i} the service interface type of P_i , R_i the names of the required services, and I_{R_i} their corresponding service interface types.
- $IConnectionPoint_{R_i}$ denotes the set of connect- and disconnect-methods provided by the connection point object for the required service named R_i and min_{R_i} , max_{R_i} are the lower limit resp. upper limit on the needed/allowed number of connections to the required service R_i .
- The plugs part lists the names (Pl_1, \dots, Pl_o) of all declared plugs together with their sets of provided and required service names. The first set in the tuple denotes the provided service names, the second set denotes the required service names.
- The constraints part lists the names (C_1, \dots, C_r) of all constraints of kind *Different Service Providers* together with their sets of required service names, C_{set}^i ($1 \leq i \leq r$).

For the plugs and constraints in the component interface the following conditions hold:

- For the reasons already mentioned on page 108 required services belonging to a constraint must not belong to a plug (constraint rule 2):

$$\forall i, 1 \leq i \leq o \text{ and } \forall j, 1 \leq j \leq r : \text{Required}(CI, Pl_i) \cap C_{set}^j = \emptyset.$$
- For the reasons mentioned below, all plugs have to be distinct from one another. That is, for two plugs Pl_k, Pl_l with $l \neq k$ the following conditions hold :
 - $\text{Provided}(CI, Pl_k) \cap \text{Provided}(CI, Pl_l) = \emptyset$
 - $\text{Required}(CI, Pl_k) \cap \text{Required}(CI, Pl_l) = \emptyset,$

where $\text{Provided}(CI, Pl_i)$ is the set of the provided service names of the plug Pl_i of the component interface CI and $\text{Required}(CI, Pl_i)$ the set of its required service names.

Reason: Assume two plugs Pl_1 and Pl_2 which have, let us say, one provided service P and one required service R in common. In addition, the two plugs have further services which are distinct from one another. Then one of the plugs can be connected to a fitting plug of another component instance thereby breaking the “integrity” of the other plug as being a unit for interconnection as defined on page 99. This situation is shown in Figure 4.12.

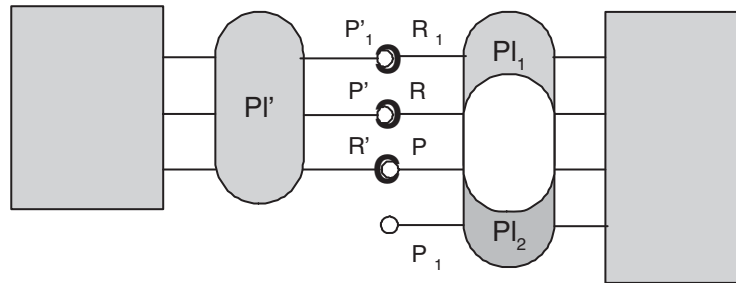


Figure 4.12: Overlapping Plugs

Here Pl_1 is connected as a whole to a fitting plug Pl' . Via R , Pl_1 is a client of the provided service P' of Pl' . Thus, via this connection between R and P' , Pl_2 is also a client of P' . Thus, as a unit of interconnection, Pl_2 would also have to be a client of P'_1 , the other provided service of Pl' . But this is obviously not true. Similarly, Pl' is no client of P_1 although it is a client of P . Thus Pl_2 and Pl' are only partly connected and thus would not agree on a common communication protocol expressed by the services involved in the protocol.

The following examples present valid component interfaces.

Example 4.1.23 (Interface of a Component used for Data Logging Purposes) *Such a component can be used by a visualization system which visualizes the received data of e.g. different sensors of one or more robot(s). The implementing component is assumed to be a JavaBean. Therefore, the service interface types are denoted as Java types.*

*The component interface is named `CIDataLogging` (see also Figure 4.17). It has two provided services (`ServiceAccess` and `LoggingOptions`) as well as two required services (`DataLogging` and `ErrorMessage`s). For provided services, only the corresponding service interface types (e.g. `ILoggingOptions`) are needed as additional information. For required services, also the types of their connection point objects (e.g. `IConnectionPoint_Error`) have to be noted as well as the lower limit (0) resp. upper limit (1) on the number of needed / allowed connections. All services except `ServiceAccess` are grouped by a plug named `LogServices`. This forces a client of `LoggingOptions` to be at the same time the receiver of the logged data and error messages and ensures e.g. that the received data and error messages correspond to the expected device selected by the client by a prior call to `selectDevice`. As there are no constraints of kind **Different Service Providers**, this part does not appear in the component interface specification.*

Java types needed as service interface types	Component interface
<pre> interface I_ServiceAccess { // returns a reference to the object // implementing the requested service Object getServiceReference (String PServiceName); // returns a reference to the connection // point object of the requested service Object getConnectionPointObject (String RServiceName); //... } interface I_LoggingOptions { // Select actual device / sensor boolean selectDevice(int deviceNumber); void startDataLogging(); void stopDataLogging(); // } interface I_DataLogging { byte[] receiveLoggedData(); // } interface IConnectionPoint_DataLogging { void connectReceiverOfLoggedData (I_DataLogging receiver); void disconnectReceiverOfLoggedData (I_DataLogging receiver); } interface I_Error { // Signals time out for returned device number int timeOut(); int DeviceNoLongerAvailable(); // ... } interface IConnectionPoint_Error { void connectErrorHandler (I_Error errorHandler); void disconnectErrorHandler (I_Error errorHandler); } </pre>	<pre> ComponentInterface CI_DataLogging { GeneralDescriptions NamingConventions = JavaType ServiceDefinitions ProvidedServices ServiceAccess : I_ServiceAccess LoggingOptions : I_LoggingOptions RequiredServices DataLogging : (I_DataLogging, IConnectionPoint_DataLogging, [0...1]) ErrorMessage: (I_Error, IConnectionPoint_Error, [0...1]) ServiceRelations Plugs LogServices = ({LoggingOptions}, {DataLogging, ErrorMessage}) } </pre>

Example 4.1.24 (Component Interface of Model) *In this example the component interface specification of the component Model introduced in the Model View Controller example (example 4.1.17) is shown.*

```

ComponentInterface CI_Model {
    GeneralDescriptions
        NamingConventions = JavaType

    ServiceDefinitions
        ProvidedServices
            ServiceAccess : I_ServiceAccess
            FileAccess     : I_FileAccess
            WriteOps       : I_TextWriteOperations
            ReadOps        : I_TextReadOperations

        RequiredServices
            TextChanges    : (I_ChangePropagation,
                           IConnectionPoint_TextChanges,
                           [0...*])

    ServiceRelations
        Plugs
            Updates        = ({ReadOps}, {TextChanges})
}

```

The provided service ReadOps and the required service TextChanges are grouped by the plug Updates. As it should be possible to connect an arbitrary number of views and controllers, there is no upper limit on the number of connections for the required service TextChanges; this is denoted by ''.*

All types of the services are already declared in examples 4.1.17 and 4.1.23. The only Java type not defined up to now is the type IConnectionPoint_TextChanges. This type specifies the connect- and disconnect-methods for the required service TextChanges and can be declared as follows:

```

interface IConnectionPoint_TextChanges {
    boolean connectTextChangeListener
        (I_ChangePropagation textChangeListener);
    boolean disconnectTextChangeListener
        (I_ChangePropagation textChangeListener);
}

```

A last example demonstrates the use of constraints.

Example 4.1.25 (Component Interface with Constraints)

A component called *OrderAdministration* administers customer and order data. To perform its task, it uses two lists, one to store and retrieve the data for customers and one to store and retrieve the data for orders. As *OrderAdministration* should not depend on a special list implementation, it declares two required services of type *I_List*, one for customers and one for orders. *I_List* declares the interface expected from a component providing a special list implementation. Every time an instance of the component *OrderAdministration* is created, it has to be connected to two **different** instances of a *List*-component. To express this requirement, a corresponding constraint called *DifferentLists* is declared in the component interface specification for *OrderAdministration*.

```
ComponentInterface CI_OrderAdministration {
  GeneralDescriptions
    NamingConventions = JavaType

  ServiceDefinitions
    ProvidedServices
      ServiceAccess      : I_ServiceAccess
      Orders             : I_Order
      Customers          : I_Customer

    RequiredServices
      OrderList          : (I_List,
                          IConnectionPoint_List,
                          [1...1])
      CustomerList       : (I_List,
                          IConnectionPoint_List,
                          [1...1])

  ServiceRelations
    Constraints
      DifferentLists     = {OrderList, CustomerList}
}
```

In this example *OrderAdministration* uses the same connection point object for both required services. The connect- and disconnect-methods therefore need an additional parameter *serviceName* which distinguishes between the different services⁷.

```
interface IConnectionPoint_List {
  void connect (I_List list, String serviceName);
  void disconnect (String serviceName);
}
```

⁷This gives more flexibility than it is possible in COM or CCM. COM only allows one connect-method for all outgoing interfaces, Advise. CCM requires distinct connect-methods for all receptacles. The name of the connect-method has to contain the name of the receptacle.

4.1.2.2 Additional Specifications supporting Automatic Connections

If assemblies are generated using tools, it should be possible to connect a required service R and a provided service P selected by a user without further user interaction. As the connection is established by calling one of the connect-methods belonging to the required service, the connect-method actually to be used has to be determined from the connection point interface. To avoid a selection process, a *default connect-method* can be determined by a declaration in the component interface specification. Every time, a connection to the required service R is to be established, this default-method is used if not otherwise stated by a user. This even simplifies composition in contexts, where no tools are used. If an assembly is just defined using a composition language as described in Section 4.2.2, interconnections between two services can be declared without the need to specify a connect-method (see Figure 4.16).

If a default-method is to be used, this method has to be declared in the component interface specification by its name and a sequence declaring its parameters. For every parameter its type is specified as well as a default value. Different parameter specifications are separated by commas. Due to technical reasons discussed later on in this thesis, only parameters of primitive data types are allowed with one exception: the parameter used to assign a reference to the service provider. This parameter has to be of the service interface type of the required service. Since the value for this parameter can not be determined in advance, it is represented by a hyphen. Similarly, a default disconnect-method has to be declared to enable automatic reconfiguration.

Thus, a declaration of a required service containing a default connect- and disconnect-method, looks as follows.

$$R : (I_R, \\ IConnectionPoint_R, \\ (\text{connect-method specification}, \\ \text{disconnect-method specification}) \\ [min_R...max_R])$$

where the method specifications consist of the method name, parameter types and values:

$$\text{methodName} (\text{type-}par_1 \text{ value-}par_1, \dots, \text{type-}par_m \text{ value-}par_m)$$

Now we give an example of a component interface specification using default connect- and disconnect-methods.

Example 4.1.26 (Default Connect- and Disconnect-Methods) *The default methods are declared for the required service SendOperations.*

```

ComponentInterface CI_Sender {
    GeneralDescriptions
        NamingConventions = JavaType

    ServiceDefinitions
        ProvidedServices
            ServiceAccess      : I_ServiceAccess
            UserRequests       : I_UserRequests
            Acknowledgements: I_Acknowledgements

        RequiredServices
            SendOperations      : (I_SendOperations,
                                IConnectionPoint_SendOperations,
                                (connectReceiver (I_SendOperations -), /* default methods */
                                 disconnectReceiver(I_SendOperations -)),
                                [1...*])

    ServiceRelations
}

interface IConnectionPoint_SendOperations {
    void connectReceiver (I_SendOperations receiver);
    void disconnectReceiver(I_SendOperations receiver);
    /* ... */
}

```

4.2 Composition

As already addressed in Sections 2.2 and 3.2.1, most of the mentioned industrial component models only provide a flat component model that is, compositions only define a web of interconnected component instances whereas interconnections are based on interfaces or on events. These models provide no means to define new components with a dedicated interface to the outer world. In contrast to these flat industrial component models, COM offers a means to compose components hierarchically by aggregation. Aggregation means, that a component may provide references to interfaces of an internal (sub)component to its clients. The clients of the aggregating “outer” component *O* do not know about the internal aggregation. They may query a reference to an interface of the aggregated component by a call to the IUnknown interface of *O* as if these interfaces were directly implemented by *O*. Unfortunately, there are several drawbacks in this approach: One of them is that components must be aware, whether they should be

used as aggregates in the future. Possible aggregates have to provide special additional features. Components missing these features can not be used as aggregates later on. Another drawback is that the means for hierarchical composition provided by COM are only designed for experienced programmers. They have to use existing programming languages to compose components, which do not support component composition concepts in a first class manner (see the description of aggregation on page 38).

Our model supports flat compositions as well as hierarchical compositions. The composition techniques available to build a web of interconnected component instances are described in the next section. In addition to interconnections based on interfaces/services, our model also allows interconnections based on plugs.

Additionally, our components may be composed hierarchically to new components of a higher level of abstraction without the need that components used as constituents must prepare themselves to be composable. It will be possible to compose components visually and to integrate components of the mentioned industrial component models (two of the main goals of this thesis). Thus even components from flat component models may be composed hierarchically.

4.2.1 Interconnections between Component Instances

There are two levels for connections between component instances: one on service level and the other one on the level of plugs.

4.2.1.1 Connections via services

In Section 4.1.1.1 we have already described, how connections via services can be established. Here, we shortly summarize this process. Let A be a component instance with a required service named R . The service interface of R is I_R and the interface declaring the connection point methods belonging to R is ICP_R . Let B be a component instance with a provided service named P which provides at least all methods declared by I_R . To connect both component instances via R and P , we first have to obtain the service object SO bound to P and the connection point object CPO belonging to R . Then we can call one of the connect-methods belonging to ICP_R on CPO passing as actual parameter for the service provider a reference to SO . The connection is established and A can now call the needed methods on the proxy belonging to R which result in calls on SO .

The service object SO can be obtained by a call to `getServiceReference` on the service access interface of B passing as parameter the name of the provided service, P . Similarly, the connection point object CPO can be obtained by a call to `getConnectionPointObject` on the service access interface of A passing as parameter the name of the required service, R . For an example please refer to example 4.1.14 on page 93.

In this context we say that a provided service P *matches* a required service R , if P declares at least all methods as provided ones which R declares to be required. Matching services are introduced more precisely in Section 4.4.1.

4.2.1.2 Connections via plugs

Connections on a higher level of abstraction can be established using plugs. A plug Pl of one component instance can be connected to a plug Pl' of another component instance, if both plugs represent the same communication protocol. That is, Pl' has as many provided services as Pl has required ones and the provided services of Pl' match the required services of Pl . The same holds for provided services of Pl and required services Pl' . In other words, both plugs have “complementary services” or “counterparts” which means that the services denote the same functionality, but have inverse roles concerning provided and required attributes/tags. Complementary services are not forced to have the same names. Two plugs with these characteristics are called to be *complementary plugs*. Complementary plugs are defined more precisely by term definition 4.4.7 in Section 4.4.

Using a tool to connect component instances, a connection via plugs can be done by a single operation from the user’s point of view. Internally, it is reduced to service connections. An algorithm deciding whether two plugs are complementary can be found in Section 4.11.1.1. Section 4.11.1.2 presents a modified version of this algorithm which automatically maps corresponding provided and required services of two complementary plugs to each other.

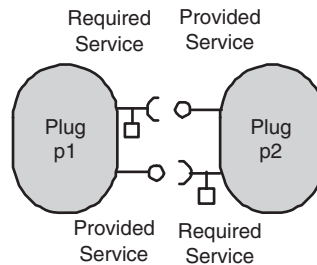


Figure 4.13: Connections via Plugs

A plug connection could be regarded as a kind of transaction. If the connection to one of the partners fails, the plug connection must be aborted. Using two plugs p1 and p2 with only one required and one provided service this would mean for example: if the required service of p1 is already connected to the provided service of p2, but the connection between the required service of p2 and the provided service of p1 fails, the previously established connection must be disconnected.

4.2.2 Hierarchical Composition

Our approach to compose industrial components hierarchically is to integrate them into our component model and to use them as atomic components (called *atomic UCM-components*). Higher level components are built using a special assembly description,

which is capable to specify hierarchically composed components (called *composite UCM-components*) in a simple way. This assembly description is referred to as *component implementation* because from a user's point of view it is like a very high level programming language.

Component interfaces for our developed component model are described by special metadata introduced in Section 4.1.2. Components of an industrial component model are integrated in our model essentially by providing these additional metadata for their component interfaces and a component implementation which simply refers to the component of the industrial component model in a suitable way. Therefore our approach allows us to describe component interfaces and component implementations uniformly for atomic UCM-components and for composite UCM-components independent of the industrial component model used for atomic UCM-components.

In the following, the component implementation for our component model is described in more detail. The component implementation refers to an existing industrial component only or it consists of a description of its aggregated component instances, their interconnections, the exported services and plugs (see below), and eventually a binding of special component implementations to component interfaces or single constituents. An aggregated component instance (part⁸) is only described by an instance name and the type of its component interface. (This corresponds to an attribute declaration for a class.) When a composite UCM-component is instantiated, for every of its parts component instances are created, too. To be as flexible as possible the parts are only typed by component interfaces. A binding of component interfaces to component implementations can explicitly be done in the `ImplementationBinding` section of the component implementation part or left to the runtime system.

Every component implementation has to specify the type of the component interface it implements. For composite UCM-components every provided service of the implemented interface (except `ServiceAccess`) must be linked to a provided service of one of the component's constituents. The linked service is called *exported service*: although it is implemented by an internal part, it becomes accessible from the outside. Similarly, every required service of the implemented component interface must be linked to a fitting required service of one constituent. Plugs of the component interface may be linked to several plugs and services of different constituents of the component. This allows one to define new communication protocols for composite components in which several constituents of the composite component work together to realize this kind of protocol.

It is not allowed to link a service of an internal constituent belonging to a plug to a service of the component interface not belonging to a plug. (Otherwise the semantics of the internal plug could be broken. See also Section 4.1.2.)

In Figure 4.14 the composite UCM-component *CPN* consists of the parts *pc₁*, *pc₂* and *pc₃* each representing an instance of type *CI₁*, *CI₂* resp. *CI₃* at runtime. The provided services of *pc₁* and *pc₂* are exported as well as the required service of *pc₃*. Also the plug

⁸The terms *part*, *constituent*, *aggregated component instance* are used synonymously.

of pc_2 is linked to the plug of the component interface and is thus exported. The required service of pc_1 is connected to the first provided service of pc_3 and the required service of pc_2 to the second provided service of pc_3 . Links of internal services to services of the component interface of the composite component are denoted by solid lines from lollipop to lollipop or from socket to socket.

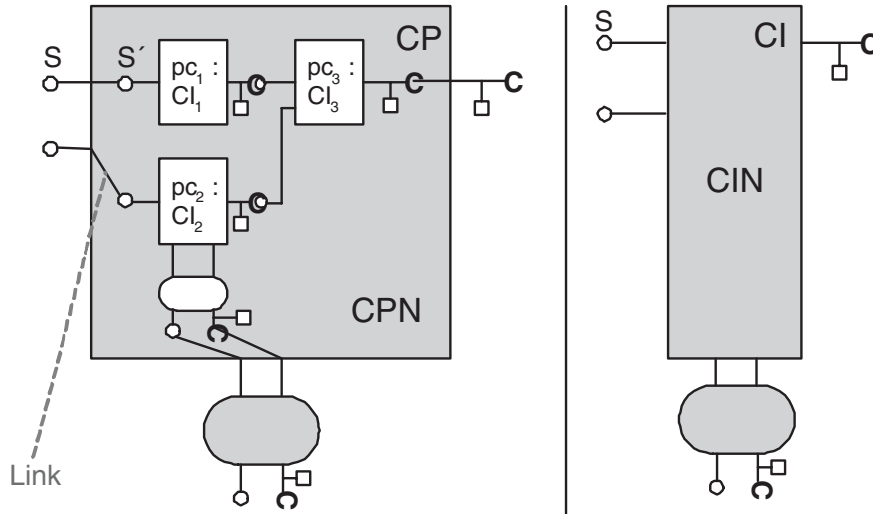


Figure 4.14: Example of a Composite UCM-Component

In the following, we give a semi-formal definition of the implementation of an atomic UCM-component and a composite UCM-component. In these definitions CPN denotes the name of the component implementation and CIN the name of the implemented component interface. The specifications concerning existing industrial component models are specialized in the corresponding subsections of Section 4.10.

Component CPN implements CIN {		(implementation of an atomic UCM-component)
GeneralDescriptions		
type	=	atomic (denotes, that CPN is implemented by an atomic UCM-component)
ComponentModel	=	CM (where $CM \in \{JavaBeans, COM, .NET, \dots\}$)
ImplementingComponent	=	IC (where IC is the name of a Java type in case of JavaBeans, a Class ID in case of COM components, a .NET type in case of .NET components etc.)
}		

Figure 4.15: Implementation for Atomic UCM-Components

For the sake of clarity the definition of the component implementation of a composite UCM-component abstracts from some details concerning interconnections (e.g.

the actual connect method used) and the handling of plugs. The missing details are described in Section 4.2.3.

Component CPN implements CIN { *(implementation of a composite UCM-component)*

```

  GeneralDescriptions
    type = composite
  Parts
     $pc_1 : CI_1$ 
    ...
     $pc_n : CI_n$ 
  InternalConnections
    RequiredServices
       $pc_{i_1}.R_{j_1} <== pc_{k_1}.P_{m_1}$ 
      ...
       $pc_{i_l}.R_{j_l} <== pc_{k_l}.P_{m_l}$ 
    Plugs
       $pc_{i_r}.Pl_{j_r} <== pc_{k_r}.Pl_{m_r}$  (in which  $Pl_{j_r}$  and  $Pl_{m_r}$  are complem. plugs (see term 4.4.7))
      ...
  Exports
    ProvidedServices
       $CIN.P_i < - - pc_{k_i}.P_{m_i}$ 
      ...
    RequiredServices
       $CIN.R_j < - - pc_{k_j}.R_{m_j}$ 
      ...
    Plugs
       $CIN.Pl_h < - - pc_{k_h}.Pl_{m_h}$ 
      ...
  ImplementationBinding
     $CI_i <<< CP_{n_i}$ 
    ...
     $pc_j <<< CP_{n_j}$ 
}
```

Figure 4.16: Implementation for Composite UCM-Components

In this description $pc_i, 1 \leq i \leq n$, denote the names of the parts (aggregated component instances at runtime), CI_i the types of their corresponding component interfaces, $pc_{i_x}.R_{m_x}$ the required service with name R_{m_x} of the aggregated internal constituent pc_{i_x} , $pc_{i_x}.P_{m_x}$ the provided service P_{m_x} of pc_{i_x} , $pc_{i_x}.Pl_{m_x}$ the plug named Pl_{m_x} of the part pc_{i_x} , $CIN.P_i$, $CIN.R_i$, $CIN.Pl_i$ the provided / required service resp. plug named P_i resp. R_i resp. Pl_i of the (implemented) component interface CIN . CP_{n_i} denote the names of the components implementing a component interface CI_i declared as component interface type of one of the parts. The binding of component implementations may be done on component interface level or on part level.

The meaning of the symbols \leq , $\leq -$ and $\leq \leq$ used in Figure 4.16 is as follows.

, \leq ' denote connections between parts via services or plugs

, $\leq -$ ' denote exported entities, i.e. which entity of the implemented component interface (left hand side) is realized by which entity of which part (internal constituent) (right hand side)

, $\leq \leq$ ' denotes, which component implementation (identified by its name) has to be used as implementation for the part specified on the left hand side or for all parts typed by the component interface specified on the left hand side depending on whether the name of a part or the name of a component interface is denoted on the left.

Thus an implementation of a composite UCM-component is like a construction plan which tells us how to build a composite, i.e. an instance of a composite component, from interconnected instances of other components.

The following example demonstrates the implementation of an atomic UCM-components as well as the implementation of a composite UCM-component.

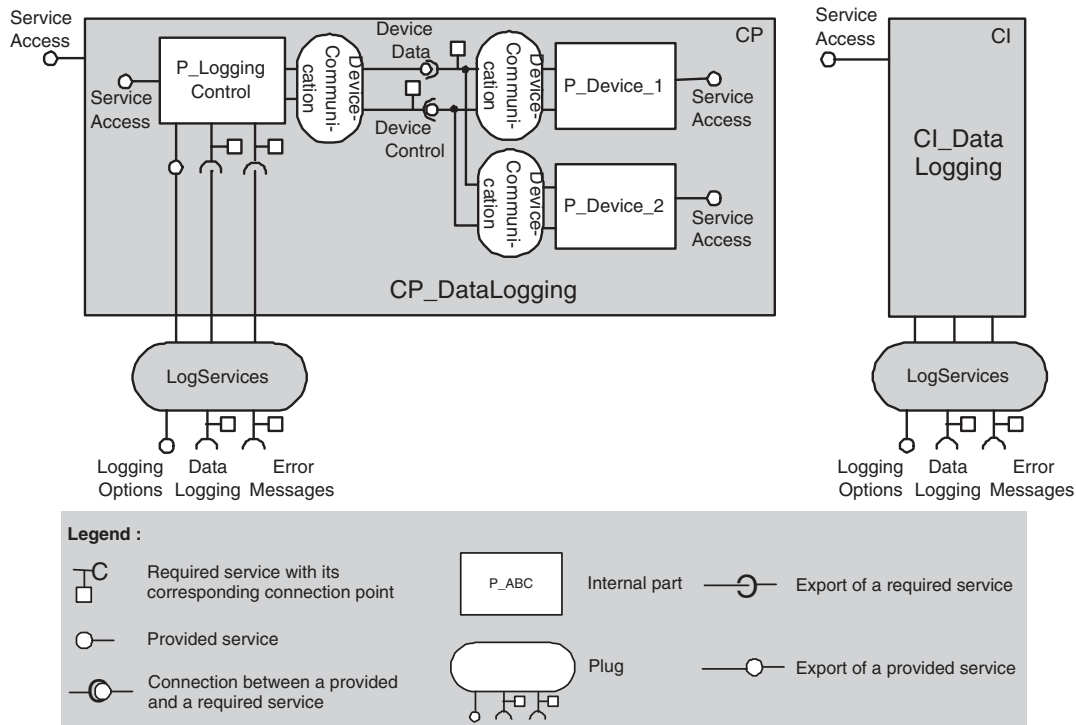


Figure 4.17: Composite UCM-Component with its Component Interface

Example 4.2.1 (Component Implementation) This example presents an implementation of a data logging component providing the component interface already described in example 4.1.23. An instance of this component consists of instances of three atomic UCM-components: One,

controlling data logging (P_LoggingControl), and two others, submitting the data of the connected devices to the controlling component. This component selects one of the connected device components according to a call of selectDevice, receives the device data (logged data and error messages) from this component, converts the received data and error messages to the format expected by I_DataLogging resp. I_Error and submits these data via the required services DataLogging resp. ErrorMessages to the outer world. The first table presents the needed Java types and component interfaces and the second one the needed component implementations.

Additionally needed Java types	Component interfaces
<pre> interface I_DeviceControl { void startDataLogging(); void stopDataLogging(); // } interface I_DeviceData { // logged data and error messages byte[] receiveDeviceData(); // } interface IConnectionPoint_DeviceData { void connectDevice(I_DeviceData deviceData); void disconnectDevice (I_DeviceData deviceData); } interface IConnectionPoint_DeviceControl{ void connectDeviceController (I_DeviceControl deviceController); void disconnectDeviceController (I_DeviceControl deviceController); } public class LoggingControl implements I_ServiceAccess, I_LoggingOptions, IConnectionPoint_DataLogging, I_DeviceData, IConnectionPoint_Error, IConnectionPoint_DeviceControl { // ... } public class ConnectionPoint_DeviceData implements IConnectionPoint_DeviceData { // ... } public class Device implements I_ServiceAccess, I_DeviceControl { // ... } </pre>	<pre> ComponentInterface CI_LoggingControl { GeneralDescriptions NamingConventions = JavaType ServiceDefinitions ProvidedServices ServiceAccess : I_ServiceAccess LoggingOptions : I_LoggingOptions DeviceData : I_DeviceData RequiredServices DataLogging : (I_DataLogging, IConnectionPoint_DataLogging, [0...1]) ErrorMessages: (I_Error, IConnectionPoint_Error, [0...1]) DeviceControl: (I_DeviceControl, IConnectionPoint_DeviceControl, [1...2]) ServiceRelations Plugs DeviceCommunication = ({DeviceData}, {DeviceControl}) } ComponentInterface CI_Device { GeneralDescriptions NamingConventions = JavaType ServiceDefinitions ProvidedServices ServiceAccess : I_ServiceAccess DeviceControl : I_DeviceControl RequiredServices DeviceData : (I_DeviceData, IConnectionPoint_DeviceData, [1...2]) ServiceRelations Plugs // Complementary Plug to 'DeviceCommunication' // of CI_LoggingControl DeviceCommunication = ({DeviceControl}, {DeviceData}) } </pre>

The next table contains the needed component implementations. `CP_DataLogging` is a composite UCM-component containing three parts: `P_LoggingControl`, `P_Device_1`, and `P_Device_2`. All entities (services, plugs) defined in a component interface are accessed by qualified names consisting of the name of the component interface or a part providing this interface and the name of the entity, e.g. `CI_DataLogging.DataLogging` or `P_Device_2.DeviceCommunication`. The atomic UCM-component `LoggingControl` (referred to by `CP_LoggingControl`) implements the connect- and disconnect-methods of all of its required services by itself whereas `Device` (referred to by `CP_Device`) uses a separate connection point object for connections to its required service `DeviceData`. A reference to this object can be obtained by a call to the method `getConnectionPointObject` of `I_ServiceAccess`.

Atomic UCM-components	Composite UCM-components
<pre> Component CP_LoggingControl implements CI_LoggingControl { GeneralDescriptions type = atomic ComponentModel = JavaBeans ImplementingComponent = LoggingControl // LoggingControl : Java type of // implementing class } Component CP_Device implements CI_Device { GeneralDescriptions type = atomic ComponentModel = JavaBeans ImplementingComponent = Device // Device : Java type of implementing class } </pre>	<pre> Component CP_DataLogging implements CI_DataLogging { GeneralDescriptions type = composite Parts P_LoggingControl : CI_LoggingControl P_Device_1 : CI_Device P_Device_2 : CI_Device InternalConnections Plugs P_LoggingControl.DeviceCommunication <== P_Device_1.DeviceCommunication P_LoggingControl.DeviceCommunication <== P_Device_2.DeviceCommunication Exports ProvidedServices CI_DataLogging.LoggingOptions <-- P_LoggingControl.LoggingOptions RequiredServices CI_DataLogging.DataLogging <-- P_LoggingControl.DataLogging CI_DataLogging.ErrorMessages <-- P_LoggingControl.ErrorMessages ImplementationBinding CI_LoggingControl <<< CP_LoggingControl CI_Device <<< CP_Device } </pre>

4.2.3 More Details on Interconnections and Exports

If the default connect-method is not used to establish a connection to a required service or the complementary services of two plugs to be connected are not named equally,

the information concerning such interconnections has to contain more details as those shown in Figure 4.16. The same holds for export information, if the equivalent services of a plug to be exported and the linked plug of the component interface are not named equally and it is not possible to derive an unambiguous mapping between the equivalent services of both plugs. Additional information on the export of a plug is also needed, if a plug of a constituent is to be composed to a greater plug of the component interface of its enclosing component. Thus, the following two sections deal with refinements needed for interconnections and for exports of plugs in such situations.

4.2.3.1 Details on Interconnections

For every pair of required and provided services R and P one needs to know the connect-method to be used to establish the connection. If the default connect-method is not used or if no default connect-method for R is specified, the actually used method including the values for its parameters have to be specified explicitly. This specification is done in the corresponding connect statement contained in the subsection 'Required-Services' of 'InternalConnections'.

Let pc and pc' denote parts of a composite UCM-component CC , R a required service of pc and P a provided service of pc' . The known notation from Figure 4.16

$$pc.R \leq== pc'.P$$

which defines an interconnection between R and P , is refined to

$$pc.R \leq== (\text{connect-method specification}) pc'.P$$

The connect-method specification consists of the method name and a sequence declaring its parameters. For every parameter its type is specified as well as the actual value:

$$pc.R \leq== (\text{methodName}(\text{type-par}_1 \text{ value-par}_1, \dots, \text{type-par}_m \text{ value-par}_m)) pc'.P$$

Only parameters of primitive data types are allowed with one exception: the parameter used to assign a reference to the service provider. This parameter has to be of the service interface type of the required service. The value for this parameter is represented by a hyphen as shown in example 4.2.2: `connect(I_List -, String OrderList)`.

Example 4.2.2 (Interconnections with Connect-Method Specification) *The component CP_SpecialOrderAdministration uses a part of type CI_OrderAdministration as declared in example 4.1.25 and two parts of type CI_List to implement an order administration using a special list implementation for its internal lists for customers and order data. To be able to properly connect the required services of its internal part P_OrderAdministration to one of the parts of type CI_List, a call to the connect-method (which is the same for both services) has to specify the name of the required service to be connected (see example 4.1.25). Therefore it is necessary to refine each connection specification for a required service of P_OrderAdministration by a connect-method specification.*

```

Component CP_SpecialOrderAdministration implements
    CI_SpecialOrderAdministration {
    GeneralDescriptions
        type = composite

    Parts
        P_OrderAdministration : CI_OrderAdministration
        P_OrderList           : CI_List
        P_CustomerList        : CI_List

    InternalConnections
        RequiredServices
            P_OrderAdministration.OrderList
                <== (connect(I_List -, String OrderList)) P_OrderList.List
            P_OrderAdministration.CustomerList
                <== (connect(I_List -, String CustomerList)) P_CustomerList.List

    Exports
        ProvidedServices
            CI_SpecialOrderAdministration.Orders
                <-- P_OrderAdministration.Orders
            CI_SpecialOrderAdministration.Customers
                <-- P_OrderAdministration.Customers

    ImplementationBinding
        CI_List <<< ...
    }

ComponentInterface CI_SpecialOrderAdministration {
    GeneralDescriptions
        NamingConventions = JavaType

    ServiceDefinitions
        ProvidedServices
            ServiceAccess : I_ServiceAccess
            Orders        : I_Order
            Customers      : I_Customer
    }

ComponentInterface CI_List {
    GeneralDescriptions
        NamingConventions = JavaType

    ServiceDefinitions
        ProvidedServices
            ServiceAccess : I_ServiceAccess
            List           : I_List
    }
}

```

For interconnections between plugs there are additional details to specify. If complementary services of two plugs are not named equally, one has to find a mapping which maps required services of one plug to their provided counterparts of the other plug. If it is not possible to derive an unambiguous mapping between required and corresponding provided services of two complementary plugs to be connected (see Section 4.11.1), the mapping has to be done explicitly.

Let pc and pc' denote parts of a composite UCM-component CC^9 , Pl a plug of pc and Pl' a complementary plug of pc' . The known notation

$$pc.Pl \leq pc'.Pl'$$

which defines an interconnection between Pl and Pl' , is refined to

$$\left\{ \begin{array}{l} pc.Pl \leq pc'.Pl' \\ pc.R_{i_1} \leq pc'.P'_{j_1} \\ \dots \\ pc.R_{i_n} \leq pc'.P'_{j_n} \\ \\ pc'.R'_{l_1} \leq pc.P_{k_1} \\ \dots \\ pc'.R'_{l_m} \leq pc.P_{k_m} \end{array} \right\}$$

R_{i_s} , $1 \leq s \leq n$, denote the required services of Pl and P'_{j_s} , $1 \leq s \leq n$, the matching provided services of Pl' . R'_{l_s} , $1 \leq s \leq m$, denote the required services of Pl' and P_{k_s} , $1 \leq s \leq m$, the matching provided services of Pl . For the sake of simplicity, the connect-method specifications are omitted.

There is no need to mark provided or required services explicitly as 'provided' or 'required'. This is due to the fact that, by convention, required services are always written on the left hand side of an interconnection specification and provided services on the right hand side.

4.2.3.2 Details on Exports of Plugs

In the component implementation for composite UCM-components described in Figure 4.16 on page 123 the linking of a plug Pl' of a part pc' to a plug Pl of the component interface CI of the encompassing composite UCM-component CC is simply denoted by $CI.Pl \leftarrow pc'.Pl'$. This notation only refers to the names of the two plugs involved in the link. The mapping on service level is omitted. This is sufficient, as far as each provided service of Pl' which should be used as the implementation of a provided service

⁹This refinement for interconnections between plugs is only described in the context of composite UCM-components as the notation ' \leq ' for interconnections is only introduced in this context. If one specifies interconnected component instances in the same way in the context of assemblies or applications, the same refinement notation can then also be used in this new context.

of Pl has the same service name or can be mapped unambiguously to its “equivalent” service of Pl . Equivalent services can e.g. be determined based on their service interface types. A similar condition has to hold for required services declared in both plugs.

If neither an unambiguous mapping nor a mapping based on name equivalence may be found, the mapping has to be done explicitly as in the case of interconnections between complementary plugs.

The known notation

$$CI.Pl < -- pc'.Pl'$$

which defines a link / export between Pl and Pl' , is refined to

$$\begin{aligned} CI.Pl < -- pc'.Pl' \quad & \{ \\ \text{ProvidedServices} \quad & \{ \\ CI.P_{i_1} < -- pc'.P'_{j_1} & \\ \dots & \\ CI.P_{i_n} < -- pc'.P'_{j_n} & \\ \} & \\ \text{RequiredServices} \quad & \{ \\ CI.R_{l_1} < -- pc'.R'_{k_1} & \\ \dots & \\ CI.R_{l_m} < -- pc'.R'_{k_m} & \\ \} & \\ \} & \end{aligned}$$

$P_{i_s}, 1 \leq s \leq n$, denote the provided services of Pl and $P'_{j_s}, 1 \leq s \leq n$, the corresponding provided services of Pl' thereby defining the mapping f from definition 4.3.17. $R_{l_s}, 1 \leq s \leq m$, denote the required services of Pl and $R'_{k_s}, 1 \leq s \leq m$, the corresponding required services of Pl' thereby defining the mapping g from definition 4.3.17.

As for export specifications, provided services as well as required services may appear on both sides of the arrow and a component may have equally named provided and required services, we have to distinguish explicitly between the export of provided services and the export of required services.

If a plug Pl of the component interface CI of a composite UCM-component CC is composed of plugs and stand-alone services of several parts of CC , we use the following notation:

$$CI.Pl < -- (\{ pc_{i_1}.P_{i_{1_1}}, \dots, pc_{i_1}.P_{i_{1_n}}, \dots, pc_{i_m}.P_{i_{m_1}}, \dots, pc_{i_m}.P_{i_{m_o}} \}, \\ \{ pc_{j_1}.R_{j_{1_1}}, \dots, pc_{j_1}.R_{j_{1_p}}, \dots, pc_{j_s}.R_{j_{s_1}}, \dots, pc_{j_s}.R_{j_{s_q}} \}, \\ \{ pc_{k_1}.Pl_{k_{1_1}}, \dots, pc_{k_1}.Pl_{k_{1_r}}, \dots, pc_{k_t}.Pl_{k_{t_1}}, \dots, pc_{k_t}.Pl_{k_{t_u}} \})$$

pc_{i_j} denote the parts involved in the link to Pl , $P_{i_{j_k}}$ the provided services, $R_{i_{j_k}}$ the required services and $Pl_{i_{j_k}}$ the plugs linked to Pl .

This notation does not yet describe the actual mapping from the services and plugs of the parts to the services grouped by Pl . Therefore this notation has to be refined to:

$$CI.Pl < -- (\{ pc_{i_1}.P_{i_{1_1}}, \dots, pc_{i_1}.P_{i_{1_n}}, \dots, pc_{i_m}.P_{i_{m_1}}, \dots, pc_{i_m}.P_{i_{m_o}} \}, \\ \{ pc_{j_1}.R_{j_{1_1}}, \dots, pc_{j_1}.R_{j_{1_p}}, \dots, pc_{j_s}.R_{j_{s_1}}, \dots, pc_{j_s}.R_{j_{s_q}} \}, \\ \{ pc_{k_1}.Pl_{k_{1_1}}, \dots, pc_{k_1}.Pl_{k_{1_r}}, \dots, pc_{k_t}.Pl_{k_{t_1}}, \dots, pc_{k_t}.Pl_{k_{t_u}} \}) \{$$

$$\begin{array}{l} \text{ProvidedServices} \quad \{ \\ CI.P'_{l_1} < -- pc_{i_1}.P_{i_{1_1}} \\ \dots \\ CI.P'_{l_w} < -- pc_{i_m}.P_{i_{m_o}} \\ \} \end{array}$$

$$\begin{array}{l} \text{RequiredServices} \quad \{ \\ CI.R'_{h_1} < -- pc_{j_1}.R_{j_{1_1}} \\ \dots \\ CI.R'_{h_x} < -- pc_{j_s}.R_{j_{s_q}} \\ \} \end{array}$$

$$\begin{array}{l} \text{Plugs} \quad \{ \\ < -- pc_{k_1}.Pl_{k_{1_1}} \{ \\ \dots \\ \} \\ \dots \\ < -- pc_{k_t}.Pl_{k_{t_u}} \{ \\ \dots \\ \} \\ \} \end{array}$$

For every mapping of an internal plug to a subset of provided and/or required services of Pl the specifications shown for simple exports at the beginning of this section have to be made. This is indicated by the dots inside the brackets for every plug of a part. For plugs, the left hand side of $< --$ is left empty. This is due to the fact that there is no entity of the component interface representing the subset of services of Pl linked to the internal plug. The mapping is done explicitly inside the brackets.

4.3 Used Type System for UCM-Components

In this section a type system for components is introduced including a special kind of subtyping. This type system especially allows one to define

- when services or plugs of two component instances can be connected to each other,
- when a composition is valid,
- when a component can be substituted by another one.

We use the following terms and conventions:

1. *Provided (CI)* and *Required (CI)* denote the sets of names of all provided resp. required services declared in a component interface *CI* **not** belonging to one of its plugs.
2. *Plugs (CI)* and *Constraints (CI)* denote the sets of names of all plugs resp. constraints defined in *CI*.
3. Let *Pl* be the name of a plug declared in *CI*, i.e. $Pl \in \text{Plugs}(CI)$. Then *Required (CI, Pl)* denotes the set of names of all required services belonging to *Pl* and *Provided (CI, Pl)* the set of names of all its provided services.
4. The following predefined tags used in type definitions are introduced to distinguish between the different elements of a component interface. *PS* is used for provided services and *RS* for required services. *PL* identifies plugs and *CO* constraints.

4.3.1 Type Definitions

In this section we introduce types for services, connection points, plugs, constraints and component interfaces. All types, except types for constraints, rely on sets of method signatures contained in service interfaces or connection point interfaces. Service interface types and connection point interface types denote named groups of method signatures. If such an interface type is a subtype of another one, the subtype contains all method signatures belonging to its supertype. In addition, it can declare additional method signatures. This is a widespread approach known from many OO-programming languages like Java, C#, C++ (pure abstract classes) or interface definition languages like MIDL and CORBA-IDL. Parameter types can be either service interface types or types of the type system used by the industrial component models chosen for implementing atomic components¹⁰. Parameter types other than service interface types are regarded

¹⁰Note that only components (atomic as well as composite) can be composed which depend on the same industrial component model.

as being atomic. They can only be checked for identity based on their type name. For service interface and connection point interface types, structural subtyping is available as described above.

Types for higher order concepts like plugs and component interfaces are essentially groupings of such service and connection point interface types with a distinction between provided and required parts. For these types, structural subtyping will be used, too. More details on this subject are given in the following subsections.

When integrating components belonging to industrial component models, their service interface types as well as the connection point types are denoted by names, e.g. Java type names in the case of JavaBeans, IIDs in case of COM etc. in their component interface description (see Section 4.10 on page 182). The decision, of whether one interface type is a subtype of another one depends on the mechanisms available for the industrial component model under consideration. Nevertheless, these decisions must conform to our requirement that subtypes comprise at least all method signatures of their super-types. The current type systems used to describe service interface types of industrial components are nominal type systems which ensure this property.

In the context of this section, CI denotes a component interface with provided services P_i and corresponding service interface types¹¹ I_{P_i} , required services R_i and corresponding service interface types I_{R_i} , connection point types $I_{ConnectionPoint R_i}$ as well as the lower and upper limit on the number of connections to R_i , \min_{R_i} and \max_{R_i} . T_{Atomic} denotes a set of types treated as atomic. Types belonging to T_{Atomic} can only be checked for identity.

Type Definition 4.3.1 (Method Type) Let $rt\ m\ (t_1\ x_1, t_2\ x_2, \dots, t_n\ x_n)$, $n \geq 0$, be an abstract syntax for a method declaration of a method named m ; rt is the method return type, t_i , $1 \leq i \leq n$, are the parameter types of the formal parameters and x_i , $1 \leq i \leq n$, are the names of the formal parameters. rt as well as t_i , $1 \leq i \leq n$, either belong to T_{Atomic} or are of an interface type as declared in type definition 4.3.2.

Then the type of method m is declared as $Mtype(m) = ((t_1, t_2, \dots, t_n), rt)$.

Type Definition 4.3.2 (Method Based Interface Type) Let I be the name of an interface declaring methods m_1, \dots, m_k , $k \geq 1$, denoted by the following abstract syntax:

$$I = \{ rt^{m_1} m_1 (t_1^{m_1} x_1^{m_1}, t_2^{m_1} x_2^{m_1}, \dots, t_{n_{m_1}}^{m_1} x_{n_{m_1}}^{m_1}), \dots, rt^{m_k} m_k (t_1^{m_k} x_1^{m_k}, t_2^{m_k} x_2^{m_k}, \dots, t_{n_{m_k}}^{m_k} x_{n_{m_k}}^{m_k}) \}.$$

Then the type of interface I is declared as $Itype(I) = \{m_1 : Mtype(m_1), \dots, m_k : Mtype(m_k)\}$ ¹². " $m_i : Mtype(m_i)$ " denotes a method named m_i with type $Mtype(m_i)$.

Type Definition 4.3.3 (Service Interface Type) Let S be the name of a service declared to be provided or required by CI and I_S the name of its service interface declaring the methods provided or required. Then $SIttype(CI, S) = Itype(I_S)$.

¹¹To simplify our description, service interface types and connection point types additionally have names. Nevertheless, subtyping is not based on these names, but instead on the type structure as declared in type definition 4.3.2.

¹²This definition is similar to the definition of an interface type given in [SC00a]. In the following, we sometimes use the notation MT^{m_i} instead of $Mtype(m_i)$ to simplify the representation.

Type Definition 4.3.4 (Provided Service Type) Let P_i be the name of a provided service declared in CI and I_P_i the name of its service interface. Then $PStype(CI, P_i) = SIttype(CI, P_i) = Itype(I_P_i)$.

Unlike other approaches which only regard the service interface type I_R as the type of the required service R , our type definition also takes into account limitations on the number of interconnections as well as the type of the connection point interface declaring the signatures of the methods needed to connect and disconnect service providers and requesters. Although the connection point methods are implemented by the component or an entity inside the component, they conceptually belong to the required service. They do not represent provided operations. In addition, the limitations on the number of connections also have to be regarded in type definitions. Assume two required services R_j and R_i with the same service interface and connection point types¹³, but different limits on the number of connections. Thus, one of these services cannot be used instead of the other one. Either one required service can not accept all service providers the other one can accept and/or one required service needs more service providers to be connected to it as the other one and can not be used in any composition, the other one is used without error. Thus we combine the service interface type, the connection point type and the limit on the number of connections into one (compound) type characterizing the required service.

Type Definition 4.3.5 (Required Service Type) Let R_i be a required service of CI , $CPTtype(CI, R_i) = Itype(IConnectionPoint_R_i)$ the connection point type of R_i and $Cardtype(CI, R_i) = (min_R_i, max_R_i)$ the type reflecting the cardinality constraints on interconnections to this required service. Then $RStype(CI, R_i) = (SIttype(CI, R_i), CPTtype(CI, R_i), Cardtype(CI, R_i))$.

In Figure 4.1 on page 86 resp. example 4.1.14 on page 93 we use Java types to denote service interface types. The following list shows some provided service types and one required service type used in this example. The component interface of component 1 which was not yet introduced in example 4.1.14 is called `CI_CustomerForm`. It declares the provided services `ConfigTextfields:I_Configuration`, `ConfigButtons:I_Configuration`, `Init:I_Init` and `ChangeNotification:I_ChangeNotification` as well as one required service `DataSetAccess:(I_CustomDataSetAccess, I_ConnectionPoint_CustomDataSetAccess, [1...1])`.

```

/***** Provided service types *****/

PStype(CI_CustomerForm, ConfigTextfields)
  = Itype(I_Configuration)
  = { setTextColor : ((java.awt.Color), void),
      getTextColor : (((), java.awt.Color)
    }

```

¹³The terms “connection point type” and “connection point interface type” are used synonymously.

```

PStype(CI_CustomerForm, ConfigButtons)
    = Itype(I_Configuration)
    = { setTextColor : ((java.awt.Color), void),
        getTextColor : (( ), java.awt.Color)
    }

PStype(CI_CustomerForm, Init)
    = Itype(I_Init)
    = { initialization : (( ), void) }

PStype(CI_CustomerForm, ChangeNotification)
    = Itype(I_ChangeNotification)
    = { dataSetChanged : ((CustomDataSet), void) }

/***** Required service type *****/

RStype(CI_CustomerForm, DataSetAccess)
    = ( Itype(I_CustomDataSetAccess),
        Itype(I_ConnectionPoint_CustomDataSetAccess),
        (1, 1) )
    = ( { getFirst : (( ), CustomDataSet),
        getNext  : (( ), CustomDataSet),
        getByID  : ((int), CustomDataSet),
        setByID  : ((CustomDataSet), void)
        },
        { connectDataSource : ((Itype(I_CustomDataSetAccess)), void),
        disconnectDataSource : (( ), void)
        },
        (1, 1)
    )
    = ( { getFirst : (( ), CustomDataSet),
        getNext  : (( ), CustomDataSet),
        getByID  : ((int), CustomDataSet),
        setByID  : ((CustomDataSet), void)
        },
        { connectDataSource : (({ getFirst : (( ), CustomDataSet),
        getNext  : (( ), CustomDataSet),
        getByID  : ((int), CustomDataSet),
        setByID  : ((CustomDataSet), void)
        })), void),
        disconnectDataSource : (( ), void)
        },
        (1, 1)
    )

```

In this example `CustomDataSet`, `int` and `void` belong to T_{Atomic} whereas the interface type names `I_CustomDataSetAccess` and `I_ConnectionPoint_CustomDataSetAccess` are mapped to their corresponding method based interface types (type definition 4.3.2).

As a plug of a component interface CI can contain several provided services having the same (service interface) type, it is not sufficient to only include the type of a provided service when defining the type of a plug. Instead, a provided service belonging to a plug is identified by a tuple containing its name P as well as its type: $(P, P\text{Type}(CI, P))$. Alternatively, one could abstract from the service names and arrange all provided services e.g. in a fixed order similar to a record type declaration. This would restrict all vendors providing a plug which is a subtype or a fitting plug for interconnection to declare the services in the same order.

For required services belonging to a plug the same arguments hold as for provided services. Because of their different roles, provided services have to be distinguished from required ones in the definition of a plug type. When one compares two plugs to decide whether one of them is a subtype of the other, one compares the types of the provided services of both plugs with each other and separately the required ones of both plugs. When one compares two plugs to decide, whether they can be connected to each other, one compares the provided services of one plug with the required ones of the other and vice versa.

We decided to distinguish provided services from required ones, by introducing a special tag, PS for provided, RS for required services. Alternatively, one could have used a binary tuple to represent the different sets where e.g. the first entry would correspond to the provided services and the second one to the required services. Using this approach, one has always to have in mind which position in the tuple was used for which kind of services. For similar reasons we introduced tags for plugs (PL) and for constraints (CO) as shown in definition 4.3.8.

Type Definition 4.3.6 (Plug Type) *Let Pl be a plug of CI . Then the type of Pl is defined by*

$$PL\text{type}(CI, Pl) = \{ (PS, \{(P, P\text{Type}(CI, P)) \mid P \in \text{Provided}(CI, Pl)\}) \} \cup \\ \{ (RS, \{(R, R\text{Type}(CI, R)) \mid R \in \text{Required}(CI, Pl)\}) \}$$

Type Definition 4.3.7 (Constraint Type) *Let C be a constraint of kind *Different Service Providers* with its constraint set C_{set} of required services of CI . Then $C\text{type}(CI, C) = C_{set}$*

The constraint `DifferentLists` from example 4.1.25 has the following type:

`Ctype(CI_OrderAdministration, DifferentLists) = {OrderList, CustomerList}.`

Most other approaches [SC00a, SC02, Sre01, Zen02, COM95, COR02] define the type of a component as the set of its provided and required service interface types only. Our type system also integrates the typing of plugs and takes into account constraints on interconnections. Not only the service interface types of required services are regarded, but also the signatures of their connection point methods as well as cardinality constraints (see definition 4.3.5). Constraints of kind *Different Service Providers* are integrated into the type of a component interface for similar reasons as the limit on the number of connections was integrated into the type of a required service. Two component interfaces may be incompatible only with respect to their constraints. Examples of such incompatibilities are shown in Figure 4.23.

The advantages of this approach become evident, if a component has to be substituted by another one. The definitions help us to decide, whether the substituting component is compatible with the old one (see Sections 4.3.2.1, 4.3.2.3) and they support an easy and safe exchange (see Section 4.8).

Type Definition 4.3.8 (Component Interface Type) *The type of a component interface CI is defined as*

$$\begin{aligned} CType(CI) = & \{ (PS, \{ (P, PStype(CI, P)) \mid P \in Provided(CI) \}) \} \cup \\ & \{ (RS, \{ (R, RStype(CI, R)) \mid R \in Required(CI) \}) \} \cup \\ & \{ (PL, \{ (Pl, PLtype(CI, Pl)) \mid Pl \in Plugs(CI) \}) \} \cup \\ & \{ (CO, \{ (C, Ctype(CI, C)) \mid C \in Constraints(CI) \}) \} \end{aligned}$$

The following two examples show the types of component interfaces already introduced in former examples. For the sake of readability, the method based interface types for service and connection point interfaces I are only denoted by $Itype(I)$.

Example 4.3.9 (Component Interface Type with Plugs) *This example presents the type of the component interface $CI_DataLogging$ shown in example 4.1.23.*

```
CType(CI_DataLogging) =
  { (PS, {(ServiceAccess, Itype(I_ServiceAccess))}),
    (RS, {}),
    (PL, {(LogServices, {(PS, {(LoggingOptions, Itype(I_LoggingOptions))}),
                          (RS, {(DataLogging, (Itype(I_DataLogging),
                                             Itype(I_ConnectionPoint_DataLogging),
                                             (0,1))),
                          (ErrorMessage, (Itype(I_Error),
                                             Itype(I_ConnectionPoint_Error),
                                             (0,1)))})})}),
    (CO, {}))
}
```

Example 4.3.10 (Component Interface Type with Constraints) *This example presents the type of the component interface $CI_OrderAdministration$ shown in example 4.1.25.*

```
CType(CI_OrderAdministration) =
  { (PS, {(ServiceAccess, Itype(I_ServiceAccess)),
          (Orders, Itype(I_Order)),
          (Customer, Itype(I_Customer))}),
    (RS, {(OrderList, (Itype(I_List), Itype(I_ConnectionPoint_List), (1,1))),
          (CustomerList, (Itype(I_List), Itype(I_ConnectionPoint_List), (1,1)))}),
    (PL, {}),
    (CO, {(DifferentLists, {OrderList, CustomerList})})
}
```

4.3.2 Subtyping

In the following sections we introduce subtyping rules for types of service interfaces, services, plugs and component interfaces. A subtype relation is denoted by “ \preceq ” including the case that both types are equal.

4.3.2.1 Subtyping of Services

We start with a natural subtype relation for method based interface types. An interface I_1 declaring methods m_1, \dots, m_n is a supertype of an interface I_2 , if I_2 contains at least all method declarations I_1 contains that is, I_2 declares methods m_1, \dots, m_k , $k \geq n$, and for every method m_i its name as well as its method type ($Mtype(m_i)$) are identical in I_1 and I_2 for $1 \leq i \leq n$.

Subtype Definition 4.3.11 (Subtyping for Method Based Interface Types) *Let I_1 and I_2 be two interfaces with their corresponding types $Itype(I_1)$ and $Itype(I_2)$ denoting sets of elements consisting of a method name and a method type as declared in type definition 4.3.2. $Itype(I_2)$ is a subtype of $Itype(I_1)$ denoted by $Itype(I_2) \preceq Itype(I_1)$, if $Itype(I_1) \subseteq Itype(I_2)$.*

Theorem 4.3.12 (Transitivity of Subtyping for Method Based Interface Types) *Let I_1, I_2 and I_3 be three interfaces with $Itype(I_1) \preceq Itype(I_2)$ and $Itype(I_2) \preceq Itype(I_3)$. Then $Itype(I_1) \preceq Itype(I_3)$ also holds.*

Proof: Theorem 4.3.12 follows directly from subtype definition 4.3.11 and the transitivity of the subset relation. \square

A client of a provided service P can refer to a provided service P' instead without problems, if P' provides all methods, P provides. This condition holds, if the service interface type of P' is a subtype of the service interface type of P .

Subtype Definition 4.3.13 (Subtypes for Provided Services) *Let P' of CI' and P of CI be two provided services with their corresponding service interface type names $I_{P'}$ and I_P and types as declared in type definition 4.3.4. $PStype(CI', P')$ is a subtype of $PStype(CI, P)$ denoted by $PStype(CI', P') \preceq PStype(CI, P)$, if $I_{P'} \preceq I_P$.*

Now assume, we have a component instance with a required service R to which one or more provided services of other component instances are connected. Every connection was established by a call to a connect-method belonging to the connection point interface of R . Let us assume that the information on which components and services are involved in the connections and on how a connection was established that is, which connect-method was used and which values were used as the actual parameters, is stored e.g. by an assembly tool in a file which represents the composition and which can be reloaded later on. We would now like to answer the question, which changes to the declaration of the required service of the component can be made, e.g. in a new

release, without invalidating the former existing connections. First, the new component must not call methods via its required service R which were not present in the former component. This condition can be satisfied, if the service interface type is a supertype of the service interface type declared for R in the old component. To be able to establish all connections to service providers as stored in the file, the new required service R has to provide all connect-methods which were available for the old one. This can be achieved by a connection point interface which is a subtype of the connection point interface used in the old component for R . As all former connections have to be established, the new required service must accept at least as many connections as the former one. Additionally, the new required service must not need connections to more service providers than the old one. Otherwise, the stored composition might not satisfy all required connections for R . These ideas are summarized in the subtype definitions 4.3.14 and 4.3.15. In these definitions, R' of CI' and R of CI are two required services with their corresponding types as declared in type definition 4.3.5.

Subtype Definition 4.3.14 (Cardinality Subtypes) *Cardtype(CI' , R') is a subtype of Cardtype(CI , R) denoted by $\text{Cardtype}(CI', R') \preceq \text{Cardtype}(CI, R)$, if $(\min_R \geq \min_R')$ and $(\max_R \leq \max_R')$.*

Subtype Definition 4.3.15 (Subtypes for Required Services) *RStype(CI' , R') is a subtype of RStype(CI , R), denoted by $\text{RStype}(CI', R') \preceq \text{RStype}(CI, R)$, if the following conditions hold:*

1. $\text{SIttype}(CI', R') \succeq \text{SIttype}(CI, R)$
2. $\text{CPtype}(CI', R') \preceq \text{CPtype}(CI, R)$
3. $\text{Cardtype}(CI', R') \preceq \text{Cardtype}(CI, R)$

Theorem 4.3.16 (Transitivity of Subtyping for Required Services) *Let R_1 of CI_1 , R_2 of CI_2 and R_3 of CI_3 be three required services with $\text{RStype}(CI_1, R_1) \preceq \text{RStype}(CI_2, R_2)$ and $\text{RStype}(CI_2, R_2) \preceq \text{RStype}(CI_3, R_3)$. Then $\text{RStype}(CI_1, R_1) \preceq \text{RStype}(CI_3, R_3)$ also holds.*

Proof: We have to show that

1. $\text{SIttype}(CI_1, R_1) \succeq \text{SIttype}(CI_3, R_3)$
2. $\text{CPtype}(CI_1, R_1) \preceq \text{CPtype}(CI_3, R_3)$
3. $\text{Cardtype}(CI_1, R_1) \preceq \text{Cardtype}(CI_3, R_3)$

Ad 1 $\text{RStype}(CI_1, R_1) \preceq \text{RStype}(CI_2, R_2) \implies \text{SIttype}(CI_1, R_1) \succeq \text{SIttype}(CI_2, R_2)$

$\text{RStype}(CI_2, R_2) \preceq \text{RStype}(CI_3, R_3) \implies \text{SIttype}(CI_2, R_2) \succeq \text{SIttype}(CI_3, R_3)$

$\implies \text{SIttype}(CI_1, R_1) \succeq \text{SIttype}(CI_3, R_3)$ (theorem 4.3.12)

(Theorem 4.3.12 can be used as $\text{SIttype}(CI_i, R_i)$, $1 \leq i \leq 3$, are method based interface types, see type definition 4.3.3).

Ad 2 $\text{RType}(CI_1, R_1) \preceq \text{RType}(CI_2, R_2) \implies \text{CType}(CI_1, R_1) \preceq \text{CType}(CI_2, R_2)$

$\text{RType}(CI_2, R_2) \preceq \text{RType}(CI_3, R_3) \implies \text{CType}(CI_2, R_2) \preceq \text{CType}(CI_3, R_3)$

$\implies \text{CType}(CI_1, R_1) \preceq \text{CType}(CI_3, R_3)$ (theorem 4.3.12)

(Theorem 4.3.12 can be used as $\text{CType}(CI_i, R_i)$, $1 \leq i \leq 3$, are method based interface types, see type definition 4.3.5).

Ad 3 $\text{RType}(CI_1, R_1) \preceq \text{RType}(CI_2, R_2) \implies \text{Cardtype}(CI_1, R_1) \preceq \text{Cardtype}(CI_2, R_2)$

$\implies \min_R_1 \leq \min_R_2 \wedge \max_R_2 \leq \max_R_1$

$\text{RType}(CI_2, R_2) \preceq \text{RType}(CI_3, R_3) \implies \text{Cardtype}(CI_2, R_2) \preceq \text{Cardtype}(CI_3, R_3)$

$\implies \min_R_2 \leq \min_R_3 \wedge \max_R_3 \leq \max_R_2$

$\implies \min_R_1 \leq \min_R_3 \wedge \max_R_3 \leq \max_R_1$

$\implies \text{Cardtype}(CI_1, R_1) \preceq \text{Cardtype}(CI_3, R_3)$

(type definition 4.3.5 and subtype definition 4.3.14)

□

4.3.2.2 Subtyping of Plugs

Now we consider the situation, where a plug Pl_2 can be used instead of a former plug Pl_1 . The contract of clients formerly connected to Pl_1 must not be broken, if they are connected to Pl_2 instead. To ensure that both sides involved in a plug-connection agree on the same communication protocol, we demand that the partner components are connected via complementary plugs Pl'_1 (term definition 4.4.7). That is, Pl'_1 has as many provided services as Pl_1 has required ones and the provided services of Pl'_1 match the required services of Pl_1 . The same holds for provided services of Pl_1 and required services Pl'_1 . In other words, both plugs have “complementary services” or “counterparts” which means that the services denote the same functionality, but have inverse roles concerning provided and required attributes/tags.

Figure 4.18 depicts the situation, where the plug Pl_2 should be used instead of the plug Pl_1 .

To ensure that the functionality expected by Pl'_1 is still available and that all former connections between matching services of both plugs Pl_1 and Pl'_1 can also be established between Pl_2 and Pl'_1 , the service types of Pl_2 have to be subtypes of the corresponding service types of Pl_1 . Here the arguments of Section 4.3.2.1 hold.

We might expect that Pl_2 can provide additional services. But as we expect plug-connections to be established using complementary plugs to ensure that both sides involved in the connection agree on the same protocol, such an additional provided service of Pl_2 would not have a counterpart in Pl'_1 . Pl'_1 does not know this new kind of communication protocol.

The following subtype definition for plugs reflects these arguments.

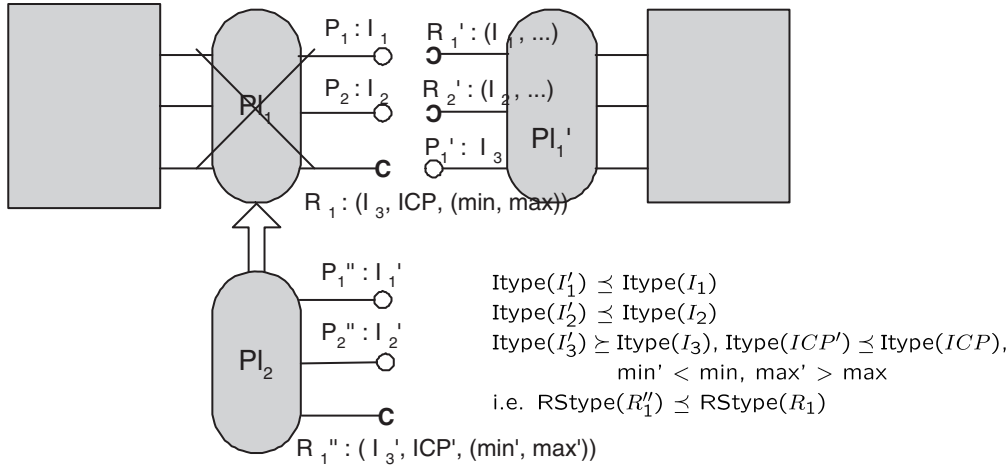


Figure 4.18: Subtyping of Plugs (IConnectionPoint is shortened to ICP)

Subtype Definition 4.3.17 (Subtypes for Plugs) Let Pl_1 of CI_1 and Pl_2 of CI_2 be two plugs with their corresponding types as declared in type definition 4.3.6. Then $PLtype(CI_2, Pl_2)$ is a subtype of $PLtype(CI_1, Pl_1)$, denoted by

$PLtype(CI_2, Pl_2) \preceq PLtype(CI_1, Pl_1)$, if the following conditions hold:

1. $\exists f : \text{Provided}(CI_1, Pl_1) \longrightarrow \text{Provided}(CI_2, Pl_2)$, f bijective with
 $\forall P \in \text{Provided}(CI_1, Pl_1) : PStype(CI_1, P) \succeq PStype(CI_2, f(P))$
2. $\exists g : \text{Required}(CI_1, Pl_1) \longrightarrow \text{Required}(CI_2, Pl_2)$, g bijective with
 $\forall R \in \text{Required}(CI_1, Pl_1) : RStype(CI_1, R) \succeq RStype(CI_2, g(R))$

The above mentioned conditions express that a plug Pl_2 may be used instead of Pl_1 , if it consists of the same number of provided and required services denoting “equivalent” services which are more specialized than those of Pl_1 . “Equivalent” services are services which provide the same functionality or which require the same functionality. A service S'_1 is more specialized than a service S_1 , if the type of S'_1 is a subtype of the type of S_1 with respect to our subtype relation.

f and g denote functions mapping the names of the provided resp. required services of Pl_1 to the names of the equivalent services of Pl_2 ¹⁴. In Figure 4.18, it is possible to use Pl_2 instead of Pl_1 , as Pl_2 satisfies all necessary conditions mentioned in subtype definition 4.3.17. f maps P_i to P_i'' ($1 \leq i \leq 2$), where P_i are the names of the provided services of Pl_1 and P_i'' the names of the provided services of Pl_2 . Similarly g maps R_1 to R_1'' , where R_1 is the name of the required service of Pl_1 and R_1'' the name of the required service of Pl_2 . Example 4.3.21 shows a valid subtype relationship between two plugs.

If a plug-connection is stored in a file, for every pair of connected provided and required services the service names are often also stored in the file (Section 4.2.3.1). If Pl_2 is used instead of Pl_1 , the service names of Pl_2 which differ from the equivalent services of Pl_1 have to be renamed in the file. If two plugs are related by a *strong* subtype relationship which retains service names, this problem does not arise.

¹⁴Note that the name of a service uniquely identifies the service itself.

Subtype Definition 4.3.18 (Strong Subtypes for Plugs) Let Pl_1 of CI_1 and Pl_2 of CI_2 be two plugs with $PLtype(CI_2, Pl_2) \preceq PLtype(CI_1, Pl_1)$. Then Pl_2 is a **strong subtype** of Pl_1 , denoted by $PLtype(CI_2, Pl_2) \preceq^s PLtype(CI_1, Pl_1)$, if both mappings f and g from subtype definition 4.3.17 can be chosen to be the identity mapping.

As intended, this subtype definition implies that a plug which is a strong subtype of another plug uses the same names for its “equivalent” required and provided services. Conversely, if a plug Pl_2 uses the same names for its provided and required services as Pl_1 and the subtype relations from definition 4.3.17 hold for every pair of equally named services of the same kind (provided / required) of both plugs that is, $PStype(CI_2, P) \preceq PStype(CI_1, P)$ for all $P \in \text{Provided}(CI_1, Pl_1)$ and $RStype(CI_2, R) \preceq RStype(CI_1, R)$ for all $R \in \text{Required}(CI_1, Pl_1)$, then $PLtype(CI_2, Pl_2) \preceq^s PLtype(CI_1, Pl_1)$. Example 4.3.21 also demonstrates strong subtyping between plugs.

Theorem 4.3.19 (Transitivity of Subtyping for Plugs) Let Pl_1 of CI_1 , Pl_2 of CI_2 and Pl_3 of CI_3 be three plugs with $PLtype(CI_1, Pl_1) \preceq PLtype(CI_2, Pl_2)$ and $PLtype(CI_2, Pl_2) \preceq PLtype(CI_3, Pl_3)$. Then $PLtype(CI_1, Pl_1) \preceq PLtype(CI_3, Pl_3)$ also holds.

Proof: We have to show that

1. $\exists f : \text{Provided}(CI_3, Pl_3) \longrightarrow \text{Provided}(CI_1, Pl_1)$, f bijective with
 $\forall P \in \text{Provided}(CI_3, Pl_3) : PStype(CI_3, P) \succeq PStype(CI_1, f(P))$
2. $\exists g : \text{Required}(CI_3, Pl_3) \longrightarrow \text{Required}(CI_1, Pl_1)$, g bijective with
 $\forall R \in \text{Required}(CI_3, Pl_3) : RStype(CI_3, R) \succeq RStype(CI_1, g(R))$

Ad 1 $PLtype(CI_1, Pl_1) \preceq PLtype(CI_2, Pl_2) \implies$

$\exists f_1 : \text{Provided}(CI_2, Pl_2) \longrightarrow \text{Provided}(CI_1, Pl_1)$, f_1 bijective with
 $\forall P \in \text{Provided}(CI_2, Pl_2) : PStype(CI_2, P) \succeq PStype(CI_1, f_1(P))$

$PLtype(CI_2, Pl_2) \preceq PLtype(CI_3, Pl_3) \implies$

$\exists f_2 : \text{Provided}(CI_3, Pl_3) \longrightarrow \text{Provided}(CI_2, Pl_2)$, f_2 bijective with
 $\forall P \in \text{Provided}(CI_3, Pl_3) : PStype(CI_3, P) \succeq PStype(CI_2, f_2(P))$

Let $f = f_1 \circ f_2 : \text{Provided}(CI_3, Pl_3) \longrightarrow \text{Provided}(CI_1, Pl_1)$

f_1 bijective $\wedge f_2$ bijective $\implies f$ bijective

Let $P \in \text{Provided}(CI_3, Pl_3)$ be chosen arbitrarily.

$PStype(CI_3, P) \succeq PStype(CI_2, f_2(P)) \wedge PStype(CI_2, f_2(P)) \succeq PStype(CI_1, f_1(f_2(P)))$
 $= PStype(CI_1, f(P))$

$\implies PStype(CI_3, P) \succeq PStype(CI_1, f(P))$ (theorem 4.3.12)

(Theorem 4.3.12 can be used as $PStype(CI_3, P)$, $PStype(CI_2, f_2(P))$ and $PStype(CI_1, f(P))$ are method based interface types, see type definitions 4.3.3 and 4.3.4).

Ad 2 The proof is similar to Ad 1. For this proof we need the transitivity of the subtype relation for required services (theorem 4.3.16). g is selected as
 $g = g_1 \circ g_2 : \text{Required}(CI_3, Pl_3) \longrightarrow \text{Required}(CI_1, Pl_1)$. □

For some of the following examples, we need the specifications of two component interfaces, `CI_LoggingControl` and `CI_LoggingControlWithNotification`. `CI_LoggingControl` was already introduced in example 4.2.1. It is repeated here to simplify the comparison of both component interfaces in the context of subtype relationships.

Example 4.3.20 (Component Interfaces for Data Logging Purposes) *We specify the component interfaces of `CI_LoggingControl` and `CI_LoggingControlWithNotification` as follows:*

<pre> ComponentInterface CI_LoggingControl { GeneralDescriptions NamingConventions = JavaType ServiceDefinitions ProvidedServices ServiceAccess : I_ServiceAccess LoggingOptions : I_LoggingOptions DeviceData : I_DeviceData RequiredServices DataLogging : (I_DataLogging, IConnectionPoint_DataLogging, [0...1]) ErrorMessage: (I_Error, IConnectionPoint_Error, [0...1]) DeviceControl: (I_DeviceControl, IConnectionPoint_DeviceControl, [1...2]) ServiceRelations Plugs DeviceCommunication = ({DeviceData}, {DeviceControl}) } </pre>	<pre> ComponentInterface CI_LoggingControlWithNotification { GeneralDescriptions NamingConventions = JavaType ServiceDefinitions ProvidedServices ServiceAccess : I_ServiceAccess LoggingOptions : I_LoggingOptionsExtended DeviceData : I_Device RequiredServices DataLogging : (I_DataLogging, IConnectionPoint_DataLogging, [0...1]) ErrorMessage : (I_Error, IConnectionPoint_Error, [0...1]) DeviceControl : (I_DeviceControl, IConnectionPoint_DeviceControl, [1...10]) DeviceManagement : (I_DeviceManagement, IConnectionPoint_DeviceManagement, [0...20]) ServiceRelations Plugs DeviceCommunication = ({DeviceData}, {DeviceControl}) } </pre>
---	---

where

```

interface I_LoggingOptionsExtended extends I_LoggingOptions { /* ... */ }

interface I_Device extends I_DeviceData { /* ... */ }

interface I_DeviceManagement { /* ... */ }

interface IConnectionPoint_DeviceManagement { /* ... */ }

```

Only the Java types not declared in example 4.2.1 are shown.

Example 4.3.21 (Subtyping of Plugs) *The type of the plug DeviceCommunication of the component interface CI.LoggingControlWithNotification from example 4.3.20 is a subtype of the type of the plug DeviceCommunication of CI.LoggingControl from the same example that is,*

$$PLtype(CI.LoggingControlWithNotification, DeviceCommunication) \preceq PLtype(CI.LoggingControl, DeviceCommunication)$$

Now we show that this relationship is true. The types of both plugs are as follows:

```
PLtype(CI.LoggingControlWithNotification, DeviceCommunication) =
  { (PS, {(DeviceData, Itype(I_Device))}),
    (RS, {(DeviceControl, (Itype(I_DeviceControl),
                          Itype(IConnectionPoint_DeviceControl), (1,10)))})
  }
```

```
PLtype(CI.LoggingControl, DeviceCommunication) =
  { (PS, {(DeviceData, Itype(I_DeviceData))}),
    (RS, {(DeviceControl, (Itype(I_DeviceControl),
                          Itype(IConnectionPoint_DeviceControl), (1,2)))})
  }
```

where

```
interface I_Device extends I_DeviceData { /* ... */ }
```

Both plugs only have one provided and one required service.

The service interface type of DeviceData of CI.LoggingControlWithNotification is Itype(I_Device). This is a subtype of Itype(I_DeviceData)¹⁵, the service interface type of DeviceData of CI.LoggingControl. Thus PStype(CI.LoggingControlWithNotification, DeviceData)

$$\preceq PStype(CI.LoggingControl, DeviceData).$$

Sltypes(CI.LoggingControlWithNotification, DeviceControl)

$$= Sltypes(CI.LoggingControl, DeviceControl),$$

CPtrype(CI.LoggingControlWithNotification, DeviceControl)

$$= CPtrype(CI.LoggingControl, DeviceControl) \text{ and}$$

Cardtype(CI.LoggingControlWithNotification, DeviceControl)

$$\preceq Cardtype(CI.LoggingControl, DeviceControl), \text{ as both lower limits are 1 and the upper limit of DeviceControl of CI.LoggingControlWithNotification is greater than the upper limit of DeviceControl of CI.LoggingControl.}$$

Thus RStype(CI.LoggingControlWithNotification, DeviceControl)

$$\preceq RStype(CI.LoggingControl, DeviceControl).$$

Therefore, using the identity as the mapping between the names of the provided services of both plugs and also between the names of the required services of both plugs, the conditions of subtype definition 4.3.17 hold. f maps DeviceData of plug DeviceCommunication of CI.LoggingControl to DeviceData of plug DeviceCommunication of CI.LoggingControlWithNotification and g maps DeviceControl of plug DeviceCommunication of CI.LoggingControl to DeviceControl of plug DeviceCommunication of CI.LoggingControlWithNotification. As f and g from subtype definition 4.3.17 are identity mappings, even strong subtyping holds:

$$PLtype(CI.LoggingControlWithNotification, DeviceCommunication) \preceq PLtype(CI.LoggingControl, DeviceCommunication).$$

¹⁵Subtyping for Java interfaces fulfils our requirement that interface subtypes contain at least all methods of their supertypes.

4.3.2.3 Subtyping of Component Interfaces

The subtype definition for component interfaces is intended to support polymorphic component instances to be used in assemblies/composite components as described in Section 4.8.1. In addition, subtyping between component interfaces should support an easy adaptation of existing components to changed requirements in new releases.

Especially, if a component interface CI_2 is a subtype of a component interface CI_1 , then every instance of a component implementing CI_1 should be substitutable by an instance of a component implementing CI_2 in any possible existing assembly (see Sections 4.2.2 and 4.8). This substitution should not affect other component instances of the assembly. Especially, no additional interconnections or exports should be introduced, no additional component instances added, existing instances, interconnections or exports should not be deleted. As shown in Section 4.8, this behavior can be guaranteed, if subtyping of component interfaces is defined as in subtype definition 4.3.22.

We motivate the different conditions listed in the subtype definition for component interfaces by explanations following immediately after this definition. The explanations are structured according to the list of conditions given in the definition and might be read in parallel.

Subtype Definition 4.3.22 (Subtypes for Component Interfaces) CI_2 is a subtype of CI_1 , denoted by

$CType(CI_2) \preceq CType(CI_1)$, if the following conditions are satisfied:

1. $\exists f: \text{Provided}(CI_1) \longrightarrow \text{Provided}(CI_2)$, f injective with
 $\forall P \in \text{Provided}(CI_1) : P\text{Type}(CI_1, P) \succeq P\text{Type}(CI_2, f(P))$
2. $\exists g: \text{Required}(CI_1) \longrightarrow \text{Required}(CI_2)$, g injective with
 $\forall R \in \text{Required}(CI_1) : R\text{Type}(CI_1, R) \succeq R\text{Type}(CI_2, g(R))$
3. $\forall R \in \text{Required}(CI_2) \setminus g(\text{Required}(CI_1)) : \min_R = 0$
4. $\exists h: \text{Plugs}(CI_1) \longrightarrow \text{Plugs}(CI_2)$, h injective with
 $\forall Pl \in \text{Plugs}(CI_1) : Pl\text{Type}(CI_1, Pl) \succeq Pl\text{Type}(CI_2, h(Pl))$
5. $\forall Pl \in \text{Plugs}(CI_2) \setminus h(\text{Plugs}(CI_1))$ and $\forall R \in \text{Required}(CI_2, Pl) : \min_R = 0$.
6. $\forall C' \in \text{Constraints}(CI_2)$ with $|g^{-1}(C'_{set} \cap g(\text{Required}(CI_1)))| \geq 2$:
 $\exists C \in \text{Constraints}(CI_1)$ with $g^{-1}(C'_{set} \cap g(\text{Required}(CI_1))) \subseteq C_{set}$

Now we give some explanations for the kind of subtyping chosen.

Condition 1 ensures that all provided services of the supertype are also available in the subtype, although the new services may have different names as well as service interface types which are subtypes of the old ones. Thus former clients of the supertype can refer to the subtype instead. The subtype may provide additional services not formerly available in its supertype. Such services are simply not used by former clients of the supertype.

Condition 2 ensures that all required services of the supertype are also available in the subtype although the new services may have different names as well as types which are subtypes of the old ones. This especially implies that the service interface types of the new required services are supertypes of the old ones. Unlike other approaches, our definition postulates that subtypes have at least as many required services as their supertypes. This is especially necessary in the context of composites as will be explained below. From Section 4.2.2 we know that a composite is built from a set of interconnected component instances having a predefined interface to the outer world. Dedicated services of the internal component instances are exported to the outside, that is, they are linked to declared services of the component interface of the composite. In Figure 4.19 the component interface of the composite is implicitly represented by the border surrounding the composite and the services sticking out of this border.

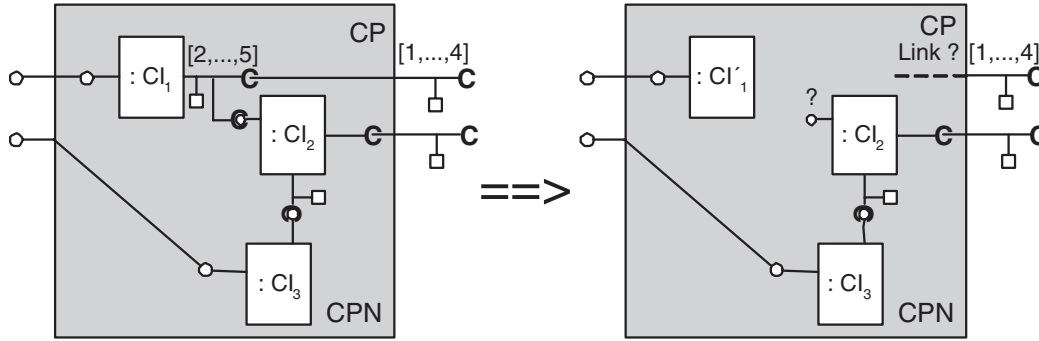


Figure 4.19: Invalidated Composite UCM-Component

An existing composite should not be invalidated, if one of its internal component instances is substituted by a component instance implementing a subtype instead of the original component interface type. That means that it must still be possible to perform all interconnections and exports originally defined for the composite UCM-component *CPN*, especially the ones the substituted component instance was involved in.

If one allows a subtype to have fewer required services than the original component interface type, this requirement could be violated as shown in Figure 4.19. In this figure, the component instance typed by CI_1 is substituted by a component instance typed by CI'_1 . In contrast to CI_1 , CI'_1 does not have any required service. Therefore, former existing connections to an instance of type CI_2 can no longer be established and it is not possible to draw a link to the first required service of the component interface of *CPN*. The missing link would result in no connection point object to be available for this required service¹⁶ and thus, it would be impossible to connect a provided service to this first required service of the composite although the component interface of *CPN* demands such a connection.

Nevertheless, components with fewer required services can be wrapped into a new component which additionally contains e.g. an extra component for each missing re-

¹⁶For a precise meaning of a link please refer to term 4.4.11 on page 158.

quired service which does not provide any functionality, but only serves to enable the establishment of former connections. This wrapped component can then be used as a valid subtype. This supports the use of components which come with their own implementation of a functionality that the supertype still required to be provided by another component. The component as such can also be installed and used to build new composite components.

Condition 3 means that CI_2 may only define additional **optional** required services. Otherwise, a mandatory required service would remain for which in actual assemblies a service provider does not exist since CI_1 does not need such a connection. This situation is illustrated by Figure 4.20.

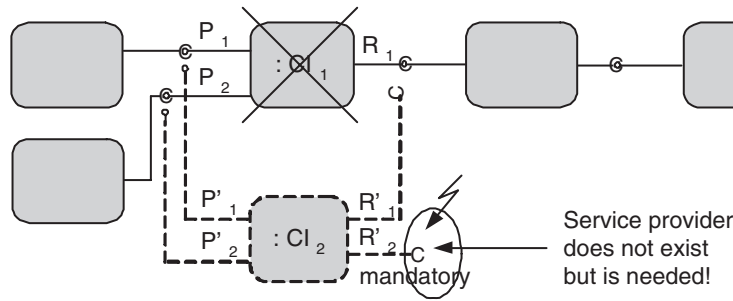


Figure 4.20: Component Interface violating Subtyping Rules for Required Services

Conditions 4 and 5 Plugs which provide functionality have to be retained in their subtypes. Here the same argument holds as for stand alone provided services. For plugs with required services, similar arguments hold as for stand alone required services. Thus a subtype must have at least as many plugs as its supertype. Additional plugs are allowed as far as these plugs only have provided services or optional required services. An additional plug must not declare additional mandatory required services.

Condition 6 expresses that existing constraints on required services of CI_1 may only be weakened on CI_2 . Preimages of required services of CI_2 not belonging to a constraint may belong to a constraint in CI_1 . If several required services of CI_2 belong to the same constraint, their preimages also have to belong to a common constraint in CI_1 .

CI_2 may have additional constraints only on optional required services not defined for CI_1 in combination with at most one service which has a preimage in CI_1 . In Figure 4.21 C_2 is weakened to C'_1 , the constraint C_1 is no longer defined on CI_2 and therefore weakened, too. Constraints are denoted by ellipses surrounding the required services belonging to the constraint.

The additional constraint C'_2 defined in CI_2 , but not in CI_1 , is only defined on the *additional* optional required services R'_7 and R'_8 . Equivalent services are not available in

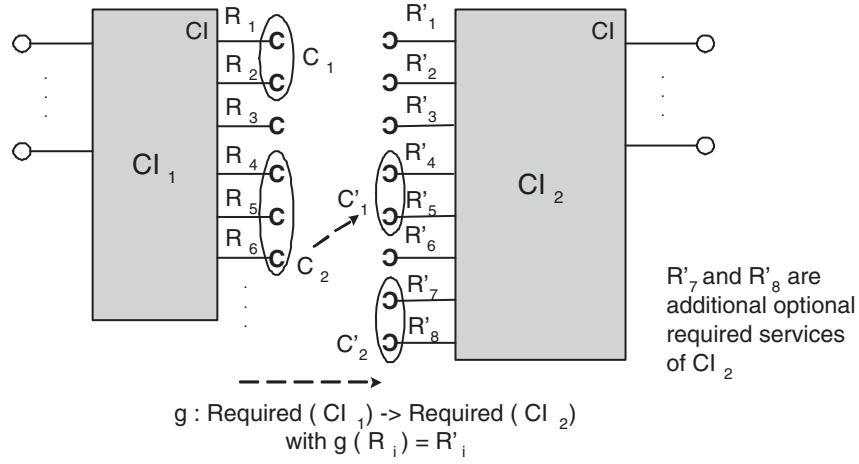


Figure 4.21: Subtyping of Constraints

CI_1 . Thus, these services can not appear in connect statements in assemblies CC which were generated for constituents typed by CI_1 . As no connections to R'_7 and R'_8 can be declared if instances of type CI_2 are used instead of instances of type CI_1 in CC , they can especially not be connected to the same service provider and thus, the constraint can not be violated.

In Figure 4.22 we give an example of a component interface CI_2 which violates condition 6. As in Figure 4.21 constraints are denoted by ellipses surrounding the required services belonging to the constraint.

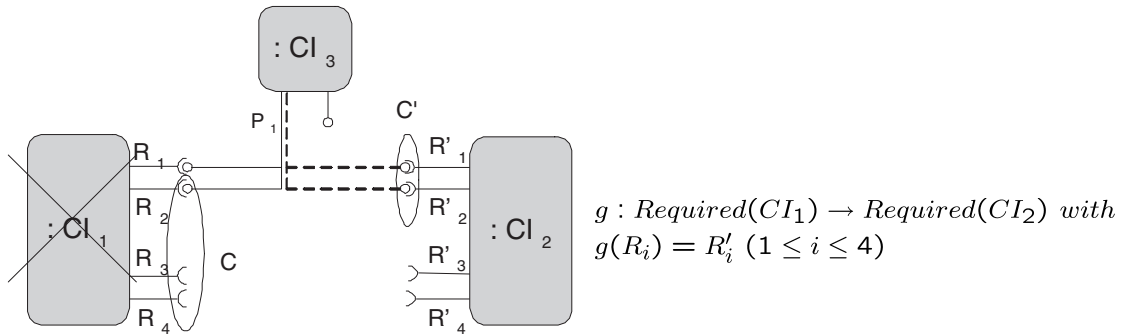


Figure 4.22: Component Interface violating Subtyping Rules for Constraints

If an instance providing CI_1 is replaced by an instance providing CI_2 , the former existing connections from P_1 to R_1 and R_2 must now be replaced by connections from P_1 to R'_1 and R'_2 . But this would violate the constraint C' which postulates that only different service providers may be connected to R'_1 and R'_2 .

Figure 4.23 shows some constraints defined in two component interfaces CI_1 and CI_2 . Circles denote required services having a counterpart with respect to the mapping g in both component interfaces. Counterparts are shown at the same position in the ellipses representing the component interfaces. Squares in CI_2 denote additional op-

tional required services without any counterpart in CI_1 . Constraints are denoted by a border surrounding the required services belonging to this constraint and are named $con\ i$, where i is a number. Constraints violating condition 6 are represented by dotted borders. Constraints in CI_1 and CI_2 which are related, are named $con\ i$ in CI_1 and $con\ i'$ in CI_2 . g maps the required services R_i of CI_1 to the equivalent required services of CI_2 denoted by R'_i .

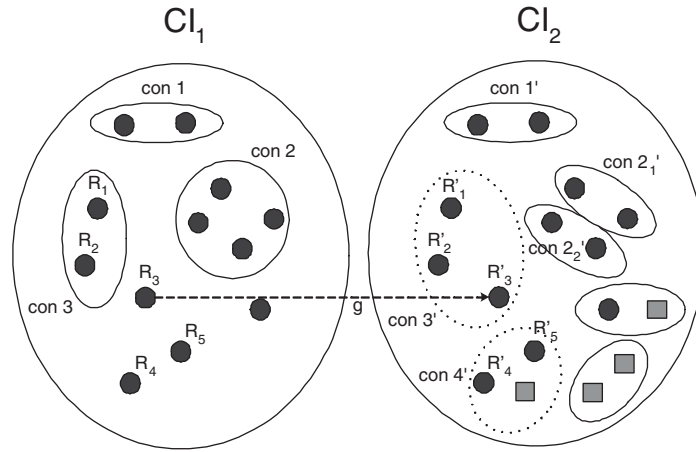


Figure 4.23: Valid and invalid Constraints concerning Subtyping

Constraint $con\ 1$ is retained by CI_2 as $con\ 1'$. $con\ 2$ is split into two weaker constraints $con\ 2'_1$ and $con\ 2'_2$. This does not cause any problems, because all service providers connected to the required services contained in $con\ 2$ are all different from one another. They thus satisfy especially the weaker conditions $con\ 2'_1$ and $con\ 2'_2$ whereas each condition demands different service providers only for two of the required services, not for all four. $con\ 3'$ is stronger than $con\ 3$ and thus violates the subtype relation. An instance of CI_2 needs different service providers to be connected to R'_1 , R'_2 and R'_3 . For an instance of CI_1 however, it is only guaranteed that R_1 and R_2 are connected to different service providers. R_3 can be connected to one of the service providers already connected to R_1 or R_2 . By similar arguments it can be shown that $con\ 4'$ can not be a valid constraint, if CI_2 is a subtype of CI_1 . The other two additional constraints in CI_2 are valid, because they only add optional required services of CI_2 not declared in CI_1 . Thus if an instance of CI_2 substitutes an instance of CI_1 in an assembly, no connections would be declared for these additional optional required services in the assembly. Therefore the constraints cannot be violated. The argumentation is the same as for the additional optional required services R'_7 and R'_8 from Figure 4.21 on page 148.

Subtyping between component interfaces as defined in subtype definition 4.3.22 does not enforce a subtype to use the same service and plug names for equivalent services and plugs. As clients have to refer to services and plugs via their names, the client code has to be adapted if instances of a subtype are used instead of instances of the supertype.

This problem can be alleviated by means of indirection which maps e.g. actual names stored in a file to variables for the names in the program code. If one wants to avoid such extra indirection or changes to the client code, a stronger subtype relation which retains names can be used.

Subtype Definition 4.3.23 (Strong Subtypes for Component Interfaces) *Let CI_1 and CI_2 be two component interfaces with $CType(CI_2) \preceq CType(CI_1)$. Then CI_2 is a **strong subtype** of CI_1 , denoted by $CType(CI_2) \preceq^s CType(CI_1)$, if the three mappings f , g and h from subtype definition 4.3.22 can be chosen to be the identity mapping. In addition, the subtype relations between plugs have to be strengthened to strong subtyping:*

$$\forall Pl \in Plugs(CI_1) : PLtype(CI_2, h(Pl)) \preceq^s PLtype(CI_1, Pl).$$

Example 4.3.24 (Subtyping of Component Interfaces) *The component interface `CI_LoggingControlWithNotification` is a subtype of `CI_LoggingControl` from example 4.3.20. The types of both component interfaces are shown below¹⁷:*

```
CType(CI_LoggingControl) =
  { (PS, {(ServiceAccess, Itype(I_ServiceAccess)),
          (LoggingOptions, Itype(I_LoggingOptions))}),
    (RS, {(DataLogging, (Itype(I_DataLogging),
                        Itype(IConnectionPoint_DataLogging),
                        (0,1))),
          (ErrorMessages, (Itype(I_Error),
                             Itype(IConnectionPoint_Error),
                             (0,1)))}),
    (PL, {(DeviceCommunication, { (PS, {(DeviceData, Itype(I_DeviceData))}),
                                   (RS, {(DeviceControl, (Itype(I_DeviceControl),
                                                             Itype(IConnectionPoint_DeviceControl),
                                                             (1,2))})})}),
    (CO, {}))
}

CType(CI_LoggingControlWithNotification) =
  { (PS, {(ServiceAccess, Itype(I_ServiceAccess)),
          (LoggingOptions, Itype(I_LoggingOptionsExtended))}),
    (RS, {(DataLogging, (Itype(I_DataLogging),
                        Itype(IConnectionPoint_DataLogging),
                        (0,1))),
          (ErrorMessages, (Itype(I_Error),
                             Itype(IConnectionPoint_Error),
                             (0,1))),
          (DeviceManagement, (Itype(I_DeviceManagement),
                                Itype(IConnectionPoint_DeviceManagement),
                                (0,20))})}),
    (PL, {(DeviceCommunication, { (PS, {(DeviceData, Itype(I_Device))}),
                                   (RS, {(DeviceControl, (Itype(I_DeviceControl),
                                                             Itype(IConnectionPoint_DeviceControl),
                                                             (1,10))})})}),
    (CO, {}))
}
```

¹⁷To increase readability, the method based interface types for the service and connection point interfaces I of the provided and required services are abbreviated by $Itype(I)$.

*The type of the provided service `LoggingOptions` of `CI_LoggingControlWithNotification` is a subtype of the type of the corresponding service in `CI_LoggingControl`. The same holds for the service `DeviceData` belonging to the plug `DeviceCommunication`. The type of the required service `DeviceControl`, also belonging to `DeviceCommunication`, is a subtype of the corresponding type in `CI_LoggingControl`, as this service allows more service providers to be connected to it. Therefore, the plug `DeviceCommunication` is a (strong) subtype of the corresponding plug in `CI_LoggingControl`. This can be verified by using the identity mapping for f and g in Def. 4.3.17. (The subtype relation between both plugs was already shown in example 4.3.21.) The new component interface has an additional **optional** required service `DeviceManagement` which signals interested 'listeners', if a new device is added or if an existing one is removed. The required services `DataLogging` and `ErrorMessage`s remain unchanged in `CI_LoggingControlWithNotification`.*

Because of our subtyping rules, `CI_LoggingControlWithNotification` is a strong subtype of `CI_LoggingControl`. The component `CP_LoggingControl` (example 4.2.1) used to instantiate all parts of `CP_DataLogging` typed by `CI_LoggingControl` (especially `P_LoggingControl`) can thus be upgraded to implement `CI_LoggingControlWithNotification` without invalidating the existing assembly description of `CP_DataLogging`. For more details please refer to Section 4.8.

4.4 Correctness of a Composition

During composition, normally two or more component instances have to be connected to each other thereby enabling the interaction needed to fulfill a common task. The composition may be either a loose assembly of interconnected component instances or an instance of a composite UCM-component. Loose assemblies use as sole composition technique composition by interconnection via services or plugs. Thus, for loose assemblies, it is only necessary to check whether the needed interconnections are allowed. If additionally hierarchical composition is used as described in the previous section, further checks to the composite have to be made which are considered below.

Provided services of the composite have to be implemented by provided services of internal constituents which is done by linking the implementing service to the corresponding service of the composite.

Every required service of the composite must be linked to a required service of an internal constituent. An interconnection to a required service of the composite then results in a connection to the corresponding internal required service. If such a link is missing, no service provider can be connected to the required service of the composite.

On the other hand, it must be possible to export mandatory required services of constituents which are not fully satisfied inside the composite by a suitable number of connections to provided services of other internal constituents. An export is done by linking the service to a required service of the composite. Thus the requirement of the internal constituent can be satisfied by a service provider connected to the linked required service of the composite. Without the link, there will be no possibility to ever establish the missing needed connections.

In the following we describe preconditions which ensure that an interconnection between a provided service of one component instance and a required service of another one can be established. We also describe what linking of services means exactly and which preconditions ensure that a service of a part may be linked to a service of the composite. Preconditions for proper interconnections between plugs and the export of plugs are added.

The conditions for valid interconnections are described in the following section. Valid exports as well as the handling of mandatory required services are described in Section 4.4.2.

4.4.1 Interconnections between Component Instances

This section only deals with connections between component instances based on services and plugs. We introduce conditions which ensure valid interconnections between component instances via services and plugs. For the following elaboration we have to clarify what it means for an object to implement an interface.

We model all runtime entities, as e.g. numbers, objects, and methods as values which are stored to locations in memory. Memory is modelled by an unbounded sequence of cells where every cell has the same fixed length e.g. in terms of bytes. A location in memory is a memory block that is, a set of consecutive cells, which is structured according to the type of entity to be stored in the location. Thus, if a location is used to hold a value of type long, the location may consist of four cells, an array of three long-values would be stored in a location consisting of twelve cells structured in blocks of four consecutive cells, each block holding one long-value etc. Locations may hold values according to their types or pointers to other locations in memory. We distinguish locations which can store values of subtypes from those allowing only values of the exact type to be stored¹⁸. We abstract from a possible indirection which is often used when handling subtypes, that is, variables which also accept values of subtypes do not hold the values as such, but hold a reference to the location in memory where the actual value is stored.

Similar to [SC00a], storing of values and pointers to other locations is modelled by a mapping S , associating locations to values or other locations that is, $S(\ell) = v$ or $S(\ell) = \ell'$ for $\ell \in Loc$, whereas Loc denotes the set of locations. $\Gamma(\ell) \equiv \tau$ denotes that the location ℓ is typed by τ and allows only values of the exact type τ to be stored. $\Gamma(\ell) = \tau$ denotes that the location ℓ is typed by τ and allows storing of values of a subtype.

If the implementation of a method m of type MT^m is loaded into memory, the method body denoted by $mbody(m, MT^m)$ is stored in a location ℓ in memory denoted by $S(\ell) = mbody(m, MT^m)$. The method body includes the list of formal parameters which means the following. If the method has n formal parameters, it refers to locations

¹⁸By this distinction we support languages like C++ or Oberon which allow values of subtypes only to be assigned to pointer types. Other kinds of variables only allow values of the exact type to be assigned. In addition, Java needs this model too as it handles value types and reference types differently.

$\ell_i^{par}, 1 \leq i \leq n$, in memory used to hold the sequence of parameter values. We define an auxiliary function *methMap* which takes as parameter a method signature and a value stored in memory. This function either returns a location in memory which holds the corresponding method body or it returns an error. We can then define what it means for an object to implement a method or even an interface. In accordance with current typed OO-programming languages we state that an object o which implements an interface I of type τ^I is itself of type τ^I .

To motivate that a function like *methMap* makes sense, look at similar functions which are even available on programming language level. For languages providing type metadata like e.g. Java or C# these functions are provided by reflection services (Sections 2.2.1.4 and 2.2.3.4). For COM, these functions are provided by type libraries (Section 2.2.2.4). E.g. in Java, an object o can be asked whether it implements a certain method e.g. by `o.getClass().getMethod(...)` and if so, its code can be executed by a call to `invoke`.

Term 4.4.1 (Object implementing a Method) *Let o be an object associated to a location ℓ in memory. That is, $\exists \ell \in Loc : S(\ell) = o$. We say o implements a method m of type MT^m , if $\exists \ell' \in Loc : methMap(m : MT^m, o) = \ell'$ and $S(\ell') = mbody(m, MT^m)$.*

Term 4.4.2 (Object implementing an Interface) *As before let $\ell \in Loc : S(\ell) = o$ and I an interface of type $Itype(I) = \tau^I$. We say o implements I , if o implements all methods m of type MT^m belonging to τ^I .*

Lemma 4.4.3 *Let I_1 and I_2 be two interfaces with corresponding types $\tau_1^I = Itype(I_1)$ and $\tau_2^I = Itype(I_2)$, $\tau_2^I \preceq \tau_1^I$. Let o be an object associated to a location ℓ in memory which implements I_2 . Then o also implements I_1 .*

Proof: As $\tau_2^I \preceq \tau_1^I$, it follows from subtype definition 4.3.11 that $\tau_1^I \subseteq \tau_2^I$. Thus, all methods belonging to I_1 also belong to I_2 . As o implements I_2 , it implements all methods belonging to I_2 and therefore especially all methods belonging to I_1 . \square

Before the body of a method can be executed, the actual parameters have to be passed to the method. This is modelled by the following process. Let $\ell_i^{par}, 1 \leq i \leq n$, be the locations in memory which are used by the method to refer to its n parameters. Let ℓ_1, \dots, ℓ_n be the locations in memory, holding the actual parameter values v_1, \dots, v_n that is, $S(\ell_i) = v_i, 1 \leq i \leq n$. Then ('call by value') parameter passing is expressed by the following mapping: $S(\ell_i^{par}) = v_i$. As above, this mapping abstracts from a possible indirection for formal parameters which allow values of subtypes to be passed that is, for which $\Gamma(\ell_i^{par}) = \tau$ for some τ . Usually such parameters do not hold the values as such, but hold a reference to the location in memory where the actual value is stored.

If $\Gamma(\ell_i^{par}) \equiv \tau_i^{par}$ for some τ_i^{par} and $\Gamma(\ell_i) \equiv \tau_i$ with $\tau_i^{par} = \tau_i$ this mapping can actually be done. The mapping can also be done, if $\Gamma(\ell_i^{par}) = \tau_i^{par}$ and $\Gamma(\ell_i) = \tau_i$ or $\Gamma(\ell_i) \equiv \tau_i$ with $\tau_i \preceq \tau_i^{par}$.

The first two cases are obvious. In the most general case $\Gamma(\ell_i^{par}) = \tau_i^{par}$ and $\Gamma(\ell_i) = \tau_i$ the mapping can be done for the following reasons. ℓ_i can hold values v of type τ^v with $\tau^v \preceq \tau_i$. Thus, if $\Gamma(\ell_i) = \tau_i \preceq \Gamma(\ell_i^{par}) = \tau_i^{par}$ and v is stored in ℓ_i , then $\tau^v \preceq \tau_i \preceq \tau_i^{par}$. Thus v can be stored to ℓ_i^{par} .¹⁹ We subsume the preconditions for valid actual parameters in the following condition.

Condition 4.4.4 (Valid Parameter Passing) *Let m be a method with signature $((t_1, \dots, t_n), rt)$ and ℓ_i^{par} , $1 \leq i \leq n$, the locations in memory for its formal parameters. Let v_i be values of type τ_i , $1 \leq i \leq n$. Then v_1, \dots, v_n can be passed as actual parameters to the formal parameters of m , if $\forall i, 1 \leq i \leq n$:*

$$\begin{cases} \tau_i = t_i, & \Gamma(\ell_i^{par}) \equiv t_i; \\ \tau_i \preceq t_i, & \Gamma(\ell_i^{par}) = t_i. \end{cases}$$

Based on these considerations, we are able to show, that two matching services can be connected to each other.

Term 4.4.5 (Matching Services) *Let P be a provided service of a component interface CI_1 and R a required service of a component interface CI_2 . Then P and R are called **matching services**, if $S\text{Itype}(CI_1, P) \preceq S\text{Itype}(CI_2, R)$ that is, the service interface type of P is a subtype of the service interface type of R .*

That a required service R of a component interface CI_2 and a provided service P of a component interface CI_1 match is a necessary precondition for an interconnection between R and P .

This is due to the fact, that, if a connection is established, the proxy belonging to R holds a reference to the service object bound to P and calls on the proxy result in calls on the service object. If $S\text{Itype}(CI_1, P) \not\preceq S\text{Itype}(CI_2, R)$, then $S\text{Itype}(CI_1, P) \not\subseteq S\text{Itype}(CI_2, R)$. Therefore, $S\text{Itype}(CI_2, R)$ contains at least one method $m : MT^m$ which is not contained in $S\text{Itype}(CI_1, P)$. Thus, if m is called on the proxy, this may lead to an error as the service object need not implement this method. It is only forced to implement the methods belonging to $S\text{Itype}(CI_1, P)$ which does not contain m . On the other hand, matching services are a sufficient precondition for an interconnection as shown below.

Theorem 4.4.6 (Valid Service Connection) *Let P be a provided service of a component interface CI_1 and R a required service of a component interface CI_2 with $S\text{Itype}(CI_1, P) \preceq S\text{Itype}(CI_2, R)$ that is, P and R are matching services. Then P can be connected to R .*

¹⁹For the type systems under consideration, we assume transitivity of the subtype relation. This is feasible as we are only interested in passing actual parameters of an interface type. For these types transitivity was proved by theorem 4.3.12.

Proof: We have to show that

1. The service object bound to P can be passed as actual parameter to every connect-method belonging to R . The corresponding formal parameter of every connect-method is the parameter of type $S\text{Itype}(CI_2, R)$ which represents a reference to a suitable service provider.
2. Every method belonging to R also belongs to P and can thus be called on the service object for P .

In the following let I_P denote the name of the service interface type of P ($P\text{Stype}(CI_1, P)$) and I_R the name of the service interface type of R ($S\text{Itype}(CI_2, R)$). SO_P denotes the service object bound to P and par_SP denotes the formal parameter of a connect-method for R used to accept a reference to a suitable service provider. Remember that we imposed the following conditions on our connect-methods: par_SP has to be of type $I\text{type}(I_R)$ and all other parameters have to be of a primitive data type. For the primitive data types, corresponding locations in memory will allow values to be stored only of the exact type. For par_SP we demand that the formal parameter declaration allows values of subtypes to be passed, too, that is, $\Gamma(\ell^{par_SP}) = I\text{type}(I_R)$. For current OO-languages this condition can be easily satisfied. For Java and C# for example this condition is satisfied in any case.

Ad 1: Term 4.1.8 states that, as SO_P is the service object bound to P , it implements I_P . Thus SO_P is of type $I\text{type}(I_P)$. As P and R are matching services, $I\text{type}(I_P) \preceq I\text{type}(I_R)$ holds. Thus, due to condition 4.4.4, SO_P can be passed as actual parameter to par_SP .

Ad 2: Using lemma 4.4.3 this follows directly from the fact that $I\text{type}(I_P) \preceq I\text{type}(I_R)$ and SO_P of type $I\text{type}(I_P)$. \square

Term 4.4.7 (Complementary Plugs) Let Pl_1 be a plug of a component interface CI_1 and Pl_2 a plug of a component interface CI_2 . Then Pl_1 and Pl_2 are called **complementary**, if the following conditions hold:

1. $\exists f : \text{Provided}(CI_1, Pl_1) \longrightarrow \text{Required}(CI_2, Pl_2)$, f is bijective with
 $\forall P \in \text{Provided}(CI_1, Pl_1) : S\text{Itype}(CI_2, f(P)) \succeq P\text{Stype}(CI_1, P)$
2. $\exists g : \text{Provided}(CI_2, Pl_2) \longrightarrow \text{Required}(CI_1, Pl_1)$, g is bijective with
 $\forall P' \in \text{Provided}(CI_2, Pl_2) : S\text{Itype}(CI_1, g(P')) \succeq P\text{Stype}(CI_2, P')$

Thus, two plugs are complementary, if every service of one plug has a one-to-one counterpart in the other plug with inverse role concerning provided and required attributes. For complementary plugs it follows that

$$\begin{aligned} |\text{Provided}(CI_1, Pl_1)| &= |\text{Required}(CI_2, Pl_2)| \quad \text{and} \\ |\text{Provided}(CI_2, Pl_2)| &= |\text{Required}(CI_1, Pl_1)| \end{aligned}$$

This fact will be used by algorithms checking whether two plugs are complementary.

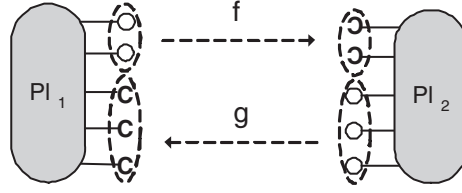


Figure 4.24: Complementary Plugs

Theorem 4.4.8 (Valid Plug Connection) *Let Pl_1 of CI_1 and Pl_2 of CI_2 be two complementary plugs. Then Pl_1 satisfies all requirements of Pl_2 and vice versa. In addition, Pl_1 and Pl_2 can be connected to each other thereby preserving the property of a plug to be a unit for interconnections.*

Proof:

1. First we have to show that $\forall R \in \text{Required}(CI_1, Pl_1) \exists P \in \text{Provided}(CI_2, Pl_2) : P$ and R are matching services.
2. Remember (page 99): “Being a unit for interconnections means the following:
 - (a) A client of a provided service belonging to a plug Pl of a component instance C is also a client of all other provided services belonging to Pl .
 - (b) The client **is forced** to be **itself** the provider for all required services of C which belong to Pl . Thus, the client has to agree on the same communication protocol as defined by Pl .”
- ad a) Let C_1 be the component instance with plug Pl_1 and C_2 the component instance with plug Pl_2 . As a provided service of Pl_2 can be connected to a required service of Pl_1 (item 1), C_1 is a client of a provided service belonging to Pl_2 via this connection. Thus C_1 has also to be a client of all other provided services of Pl_2 . If we can show that for every provided service of Pl_2 there exists a matching required service of Pl_1 , these services can be connected and thus the first condition is satisfied.
- ad b) The second condition tells us that C_1 has to provide all services which are required by C_2 via Pl_2 . If we can show that for every required service of Pl_2 there exists a matching provided service of Pl_1 , the second condition is also satisfied.
3. Items 1 and 2 have also to hold for Pl_1 and Pl_2 with inverse roles .

Ad 1: Let $R \in \text{Required}(CI_1, Pl_1)$. Item 2 from term 4.4.7 ensures that $g^{-1}(R) \in \text{Provided}(CI_2, Pl_2)$ with $\text{SType}(CI_1, R) \succeq \text{PType}(CI_2, g^{-1}(R))$. Thus $g^{-1}(R)$ and R are matching services.

Ad 2: First we show that $\forall P \in \text{Provided}(CI_2, Pl_2) \exists R \in \text{Required}(CI_1, Pl_1) : P$ and R are matching services.

Let $P \in \text{Provided}(CI_2, Pl_2)$. Then P and $R = g(P) \in \text{Required}(CI_1, Pl_1)$ are matching services. This follows directly from item 2, term 4.4.7.

Now we show that $\forall R \in \text{Required}(CI_2, Pl_2) \exists P \in \text{Provided}(CI_1, Pl_1) : P$ and R are matching services.

For $R \in \text{Required}(CI_2, Pl_2)$, $P = f^{-1}(R) \in \text{Provided}(CI_1, Pl_1)$ is a matching provided service. This follows from item 1, term 4.4.7 similar to Ad 1.

Ad 3: Item 3 can be shown using the same arguments as in Ad 1 and Ad 2. \square

A single connection between a provided service P of a component interface CI_1 and a required service R of a component interface CI_2 ($R \leq P$) is only allowed, if neither P nor R belong to a plug.

4.4.2 Export of Services and Plugs

A component implementation CPN must be checked for consistency with its declared component interface CIN . Every entity declared in the component interface (except ServiceAccess) **must** be *linked* to one or more entities of constituents used in the component implementation.

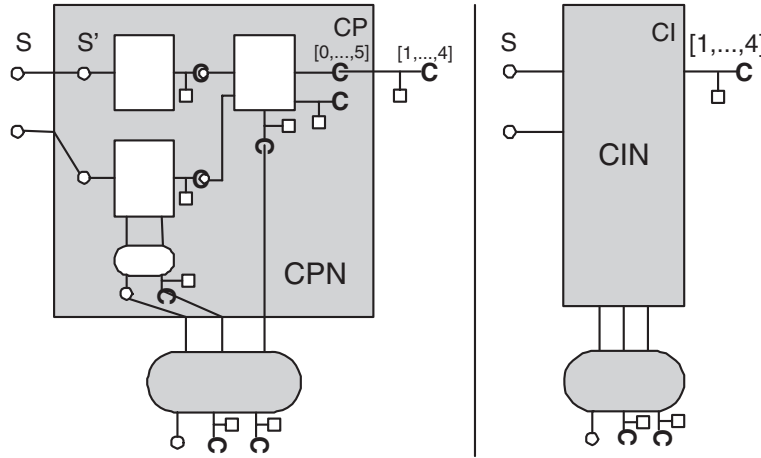


Figure 4.25: Exports

Before being able to define when linking is possible, we have to explain what a link means.

Term 4.4.9 (Link between Provided Services) *If a provided service S' of an internal constituent of a composite UCM-component CPN is linked to a provided service S of the component interface CIN implemented by CPN , then the service object bound to S' is also bound to S . That is, S and S' use the same service object.*

Theorem 4.4.10 (Valid Service Export for Provided Services) *Let S be a provided service of the component interface CIN implemented by a composite UCM-component CPN and S' be a provided service of an internal constituent of CPN typed by CI_{part} with $PStype(CIN, S) \succeq PStype(CI_{part}, S')$. Then S' can be linked to S that is, S' provides a valid implementation for S via its service object.*

Proof: We have to show that the service object bound to S implements all methods declared in the service interface type of S .

Let $PStype(CIN, S) = Itype(I_S)$ be the service interface type of S and $PStype(CI_{part}, S') = Itype(I_{S'})$ the service interface type of S' . As the service object $SO_{S'}$ bound to S' implements $I_{S'}$ and $Itype(I_{S'}) \preceq Itype(I_S)$, $SO_{S'}$ also implements I_S (lemma 4.4.3). By linking S' to S , $SO_{S'}$ is also bound to S i.e. $SO_{S'}$ is used as the service object for S . As it implements I_S , it is a valid service object for S . \square

$PStype(CIN, S) \succeq PStype(CI_{part}, S')$ is also a necessary precondition for a possible link between S and S' for the following reasons.

A client of the provided service S assumes that it can call all methods belonging to $PStype(CIN, S)$ on the service object bound to S . If S' is linked to S , then the client obtains a reference to the service object bound to S' . If $PStype(CI_{part}, S') \not\subseteq PStype(CIN, S)$, then $PStype(CI_{part}, S') \not\supseteq PStype(CIN, S)$. Therefore, $PStype(CIN, S)$ contains at least one method $m : MT^m$ which is not contained in $PStype(CI_{part}, S')$. Thus, if m is called on the service object bound to S' , this may lead to an error as the service object need not implement this method. It is only forced to implement the methods belonging to $PStype(CI_{part}, S')$ which does not contain m .

Term 4.4.11 (Link between Required Services) *If a required service S' of an internal constituent of a composite UCM-component CPN is linked to a required service S of the component interface CIN implemented by CPN , then S and S' use the same connection point object and proxy to store a reference to a suitable service object. That is, calls on the proxy for S' result in a call to the service object bound to a provided service P connected to S .*

Theorem 4.4.12 (Valid Service Export for Required Services) *Let S be a required service of the component interface CIN implemented by a composite UCM-component CPN and S' be a required service of one internal constituent of CPN typed by CI_{part} with $RStype(CIN, S) \succeq RStype(CI_{part}, S')$. Then S' can be linked to S that is, if the requirements of S are satisfied, the requirements of S' are also satisfied and the maximum number of connections allowed for S can be established without violating the maximum number of connections for S' .*

Proof: We have to show that

1. All suitable service providers for S are also suitable service providers for S' . In more detail: If sp of type CI_{sp} is a suitable service provider for S with a provided service P which matches S , then P and S' are also matching services.

2. The connection point object for S' also implements all connection point methods of S and can thus be used as a connection point object for S . Especially, every connect-method used to establish a connection between P and S is also available for S' .
3. After establishing a connection between P and S , the service object bound to P is referenced by the proxy of S' and thus serves as an implementation for the methods belonging to S' .
4. If the minimum number of connections required by S is established, the minimum number of connections required by S' is also established and thus all mandatory connections of S' are satisfied.
5. The maximum number of connections allowed for S does not exceed the maximum number of connections allowed for S' .

Ad 1: Let P be a provided service of CI_{sp} such that P and S match. Let $Itype(I_P) = PStype(CI_{sp}, P)$ denote the service interface type of P , $Itype(I_S) = SIttype(CIN, S)$ the service interface type of S and $Itype(I_{S'}) = SIttype(CI_{part}, S')$ the service interface type of S' .

The definition of matching services 4.4.5 implies that $Itype(I_P) \preceq Itype(I_S)$. The subtype relation $RStype(CI_{part}, S') \preceq RStype(CIN, S)$ implies that $Itype(I_S) \preceq Itype(I_{S'})$ (see subtype definition 4.3.15). Thus, as the subtyping relation for method based interface types is transitive (theorem 4.3.12), $Itype(I_P) \preceq Itype(I_{S'})$. Therefore P and S' are matching services.

Ad 2: Let $Itype(ICP_S) = CPtype(CIN, S)$ be the connection point type of S and $Itype(ICP_{S'}) = CPtype(CI_{part}, S')$ the connection point type of S' . As $RStype(CI_{part}, S') \preceq RStype(CIN, S)$, $Itype(ICP_{S'}) \preceq Itype(ICP_S)$ (see subtype definition 4.3.15). The connection point object for S' implements $ICP_{S'}$ and thus especially ICP_S . Therefore, this object can also be used as a connection point object for S .

Ad 3: Linking S' to S results in the same connection point object and proxy used for S and S' . Thus, if a provided service P is connected to S , the service object bound to P is stored in the proxy for S which is the same as for S' . Thus, connections to S result in connections to S' .

Ad 4: As $RStype(CI_{part}, S') \preceq RStype(CIN, S)$, $Cardtype(CI_{part}, S') \preceq Cardtype(CIN, S)$. That is, $min_{S'} \leq min_S$ and $max_{S'} \geq max_S$. If min_S - connections are established for S , min_S - connections are established for S' , because connections to S result in connections to S' (item 3). As $min_{S'} \leq min_S$, all needed connections for S' are established.

Ad 5: Follows directly from $max_S \leq max_{S'}$. □

$R\text{Type}(CIN, S) \succeq R\text{Type}(CI_{part}, S')$ is also a necessary precondition for a possible link between the required services S and S' , similar to provided services. A proof has to consider the service interface types, the connection point types and the cardinality types of S and S' .

For the linking of plugs, the following conditions have to hold. A service of an internal plug may **not** be linked to a service of CIN not belonging to a plug. Internal plugs may only be linked as a whole to a plug of the “same size” or they may be composed to yield a “greater” plug as discussed below.

Term 4.4.13 (Partially linked Plugs) *Let Pl be a plug of the component interface CIN implemented by a composite UCM-component CPN and Pl' be a plug of an internal constituent of CPN typed by CI_{part} . Then Pl' and Pl are said to be **partially linked** if*

1. $\exists f : \text{Provided}(CI_{part}, Pl') \longrightarrow \text{Provided}(CIN, Pl)$, f injective with
 $\forall P \in \text{Provided}(CI_{part}, Pl') : P$ is linked to $f(P)$
2. $\exists g : \text{Required}(CI_{part}, Pl') \longrightarrow \text{Required}(CIN, Pl)$, g injective with
 $\forall R \in \text{Required}(CI_{part}, Pl') : R$ is linked to $g(R)$

That is, two plugs Pl and Pl' can be partially linked, if every service S' belonging to Pl' can be linked to a corresponding service S of Pl of the same kind (provided or required) and corresponding services can be uniquely determined.

Term 4.4.14 (Fully linked Plugs) *Let Pl be a plug of the component interface CIN implemented by a composite UCM-component CPN and Pl' be a plug of an internal constituent of CPN typed by CI_{part} . Then Pl' and Pl are said to be **fully linked**, if*

1. $\exists f : \text{Provided}(CI_{part}, Pl') \longrightarrow \text{Provided}(CIN, Pl)$, f bijective with
 $\forall P \in \text{Provided}(CI_{part}, Pl') : P$ is linked to $f(P)$
2. $\exists g : \text{Required}(CI_{part}, Pl') \longrightarrow \text{Required}(CIN, Pl)$, g bijective with
 $\forall R \in \text{Required}(CI_{part}, Pl') : R$ is linked to $g(R)$

That is, two plugs Pl and Pl' can be fully linked, if they can be partially linked and both plugs have the same number of provided services as well as the same number of required services.

Term 4.4.15 (Linked Plugs) *Let Pl be a plug of the component interface CIN implemented by a composite UCM-component CPN and Pl' be a plug of an internal constituent of CPN typed by CI_{part} . Then Pl' and Pl are said to be **linked**, if they are fully or partially linked.*

Theorem 4.4.16 (Valid Plug Export) *Let Pl be a plug of the component interface CIN implemented by a composite UCM-component CPN and Pl' be a plug of an internal constituent of CPN typed by CI_{part} with $PL\text{type}(CIN, Pl) \succeq PL\text{type}(CI_{part}, Pl')$. Then Pl' and Pl can be fully linked.*

Proof: $\text{PLtype}(CI_{part}, Pl') \preceq \text{PLtype}(CIN, Pl) \iff (\text{subtype definition 4.3.17})$

1. $\exists f : \text{Provided}(CIN, Pl) \longrightarrow \text{Provided}(CI_{part}, Pl')$, f bijective with
 $\forall P \in \text{Provided}(CIN, Pl) : \text{PStype}(CIN, P) \succeq \text{PStype}(CI_{part}, f(P))$
2. $\exists g : \text{Required}(CIN, Pl) \longrightarrow \text{Required}(CI_{part}, Pl')$, g bijective with
 $\forall R \in \text{Required}(CIN, Pl) : \text{RStype}(CIN, R) \succeq \text{RStype}(CI_{part}, g(R))$

$\implies \forall P' \in \text{Provided}(CI_{part}, Pl') : P'$ can be linked to $f^{-1}(P')$ (theorem 4.4.10) \wedge
 $\forall R' \in \text{Required}(CI_{part}, Pl') : R'$ can be linked to $g^{-1}(R')$ (theorem 4.4.12) \square

Theorem 4.4.17 (Valid Plug Composition) *Let Pl be a plug of the component interface CIN implemented by a composite UCM-component CPN and Pl' be a plug of an internal constituent of CPN typed by CI_{part} . Let $PS' \subseteq \text{Provided}(CIN, Pl)$ and $RS' \subseteq \text{Required}(CIN, Pl)$ be subsets of the services belonging to Pl and thus $Pl'' = (PS', RS')$ a “subplug” of Pl . Let $\text{PLtype}(CI_{part}, Pl') \preceq \text{PLtype}(CIN, Pl'')$. Then Pl' and Pl can be partially linked.*

Proof: Follows directly from theorem 4.4.16 and term definition 4.4.13. \square

If all entities of CIN are completely linked to internal entities, no internal “open mandatory required service” may remain, which is not linked to an entity of CIN . “Open mandatory required service” means, that the lower limit on the number of connections is not already reached that is, needed connections are still missing. If such required service is not exported, there will be no possibility to ever establish these needed connections.

Following conditions have to hold with respect to constraints:

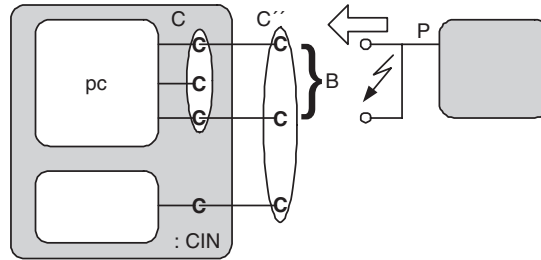


Figure 4.26: Exported Constraints

Let C be a constraint of kind *Different Services Providers* with its constraint set C_{set} of required services declared for one internal constituent pc of a composite UCM-component with component interface CIN . Let $C'_{set} \subseteq C_{set}$ be a subset of required services of pc which are linked to required services of CIN forming a set B . Then in CIN there must exist a constraint C'' with constraint set C''_{set} for which $B \subseteq C''_{set}$, if B contains at least two elements. Otherwise, as C is not known in the context of CIN , the required services of B may be connected to a common service provider which would result in this same service provider to be connected to the required services of pc contained in $C'_{set} \subseteq C_{set}$ (see term 4.4.11). This would violate the constraint C of pc .

4.5 Component Lookup

When composing new components or applications from existing ones, one needs a pool of available component implementations from which components can be chosen. Such a pool can be realized by storing all components into a directory which is made known to a tool e.g. by a configuration file. Several directories can be used to provide different pools which may e.g. contain different versions of the same components. These component pools have to be available at runtime, too, to allow composite UCM-components to be instantiated. When storing a component to a pool, also its component interface which is described in a file separate from the component implementation file, is stored in a corresponding component interface pool. We refer to component interfaces belonging to a pool as *registered component interfaces*.

As in our composite UCM-components parts are only typed by component interfaces and a composite UCM-component is not forced to declare the component implementations to be used for its parts, we may have a problem if we want to instantiate a composite UCM-component. Therefore, we need a means to map component interfaces to component implementations to be used for instantiation.

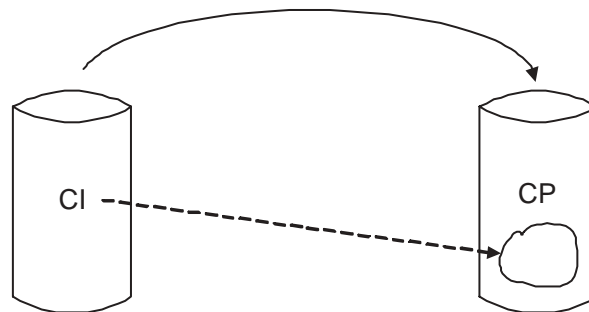


Figure 4.27: Pool of Component Interfaces and Component Implementations

For every registered component interface CI , the map provides the set of components implementing CI . This map is called *implementation registry*. Using the implementation registry, the runtime system can look for valid component implementations for every component interface typing a part in a composite UCM-component at component instantiation time. This approach has the advantage, that components implementing a component interface CI can be replaced by others implementing the same interface without affecting the implementations of composite UCM-components which use this interface as type for one of their parts.

Every time, a new component is deployed that is, is added to the component pool, the implementation registry has to be updated. The same holds, if components are removed from the pool. In this case, the administrator can instruct the runtime system to use another component as valid substitution for the removed component. This can be achieved by an additional map, called *substitution registry* which redirects old component implementations to new component implementations.

This registry is useful for composite UCM-components which internally specify the components to be used as implementation for one of their parts. Such components are not as flexible as components which specify their parts by component interfaces only. Assume, a component CP_{old} should be replaced by a component CP_{new} . Then CP_{old} will be removed from the component pool and CP_{new} will be deployed instead. If CC is a composite UCM-component which refers to CP_{old} internally by an explicit implementation binding, CC can no longer be instantiated. However, if CP_{new} is registered as a valid substitution for CP_{old} in the substitution registry, the runtime system behaves as follows. Every time an instance of CP_{old} has to be created, the runtime system will search for the old implementation. Since this implementation is no longer available, it scans the substitution registry to find a component which can be used instead. In our case, CP_{new} will be found. Thus, in all places, where CP_{old} occurs in CC , an instance of CP_{new} is created.

4.6 Instantiation of Composite UCM-Components

In consensus with other academic approaches and with the industrial component models we integrate into our model, we assume components which can be instantiated. In the following we describe the process of instantiating a composite UCM-component.

A composite UCM-component CC is instantiated by first creating instances for all of its parts and afterwards connecting all parts as declared in the section 'InternalConnections' of its component implementation. Exports are realized by a special implementation of the service access interface of the composite UCM-component which delegates a call to `getServiceReference` or `getConnectionPointObject` to the service access interface of the part exporting the requested service. The delegation process terminates on an instance of an atomic UCM-component.

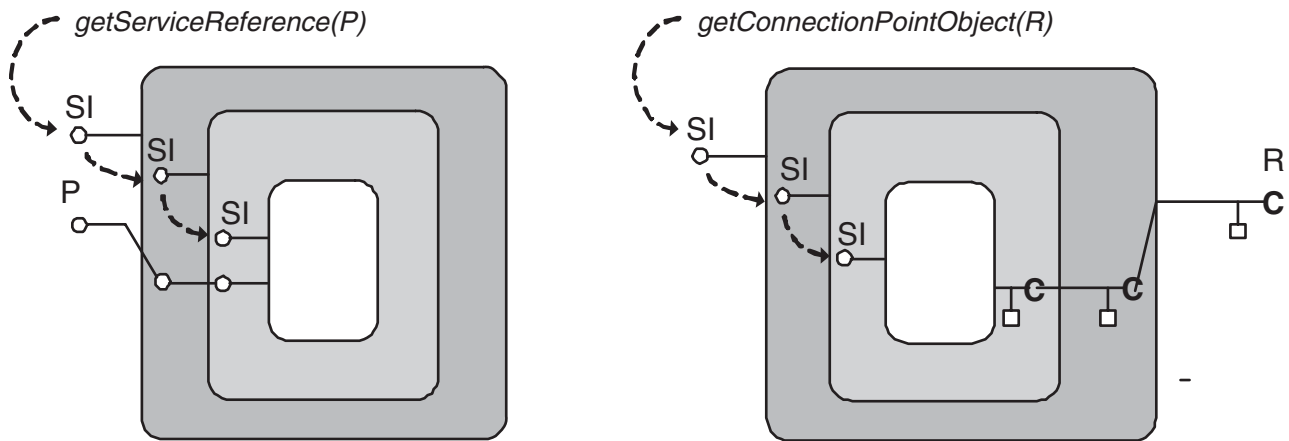


Figure 4.28: Nested Instances of Composite UCM-Components

In Figure 4.28 the service access interface is denoted by SI .

To be able to instantiate a part p , a component implementing the component interface CI which types the part has to be determined. We have to distinguish the following situations:

- In the section ‘ImplementationBinding’ of the composite UCM-component CC , CI or p is bound to a component implementation CP that is, $CI \lll CP$ or $p \lll CP$ ²⁰.
- Neither CI nor p are bound to a special component implementation.

In the first case, an instance of CP will be created for p , if CP still exists in the component pool. Otherwise, the runtime system searches the substitution registry for a valid substitution. If there is no valid substitution, instantiation is aborted. Otherwise, an instance of the substituting component is created. If instantiation does not succeed, the instantiation of CC is aborted.

In the second case the runtime system has to determine a fitting component implementation for p . For this purpose it searches the implementation registry for the components implementing CI . If CI is not registered or no component implementing CI can be found, instantiation of p fails and instantiation of CC is aborted. If CC can be successfully instantiated, one obtains a reference to its service access interface.

4.7 Helper Components

There are several situations, where the use of special helper components can simplify the composition process significantly and make the use of components more secure. Two such components, a delegator and a multiplexer component are described in this section. Their intended use and the problems they solve are described in the following.

A *delegator component* is a component having only one provided and one required service. The service interface types of the provided and required service are identical. If a client calls methods on an instance of the delegator component, the instance delegates the calls to the proxy belonging to its required service. Therefore, before being able to call the provided methods, a suitable service provider has to be connected to the required service of the instance of the delegator component. The lower and upper limit on the number of connections to the required service is one. That is, exactly one service provider can be connected.

A delegator component can be used to simplify compositions composing several constituents all having a required service of the same type exported by the composite. In the following, the composite is denoted by C . The semantics of C could require that

²⁰Remember: “ $CI \lll CP$ ” denotes that CP is used to instantiate all parts in the enclosing UCM-component which are typed by CI . “ $p \lll CP$ ” denotes that CP is used to instantiate p only (see Figure 4.16).

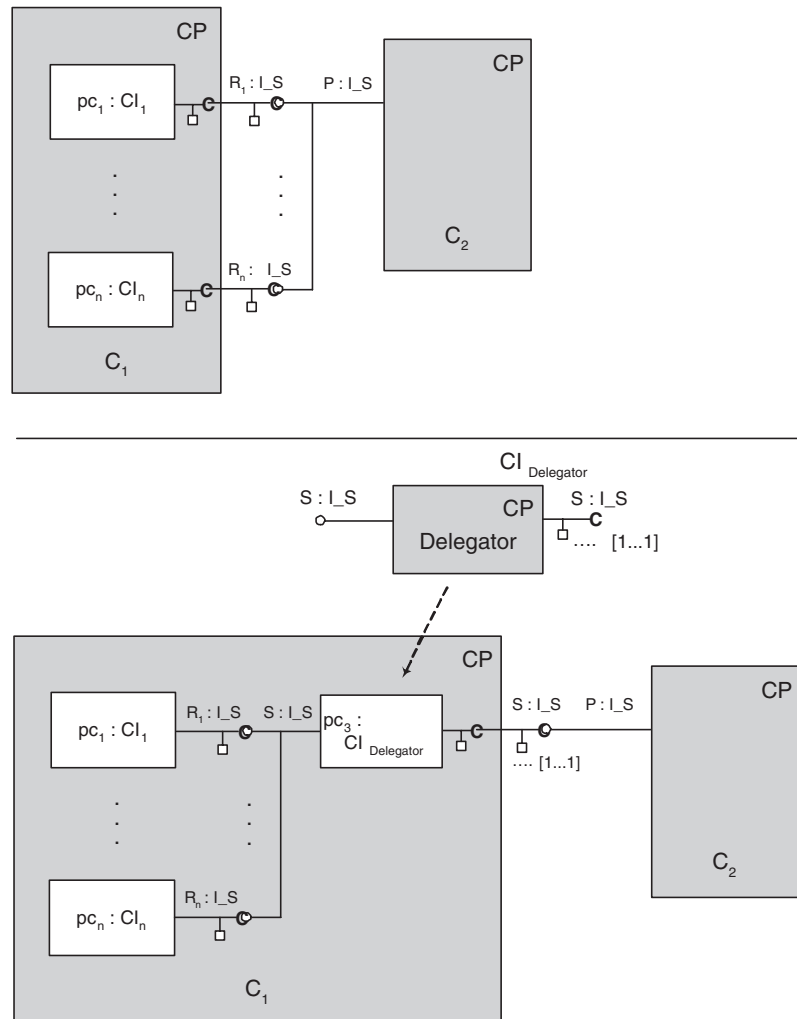


Figure 4.29: Composite UCM-Component with integrated Delegator Component

all these exported required services have to be connected to the same service provider (see Figure 4.29). Thus, other compositions using an instance of C , have to know about this semantics. Additionally, they have to establish all connections to these required services. Using instances of C in other compositions could be significantly simplified, if C is modified by using an additional delegator component in its implementation.

All required services, formerly exported, are now connected to an instance of the same delegator component. Instead of the formerly exported required services only the required service of the instance of the delegator component is exported. Thus the semantics of C is satisfied internally and other compositions using instances of C only have to establish one connection to the exported required service of the delegator. This situation is shown in Figure 4.29. In this figure C corresponds to C_1 .

Another useful helper component is a *multiplexer component*. A multiplexer compo-

nent is a component, having one provided and one required service only. As in the case of the delegator component, both service interface types are identical. But in contrast to the delegator component, the multiplexer component does not define an upper limit on the number of connections to the required service. A method called on its provided service results in a call to the same method on all service providers connected to the required service of the multiplexer.

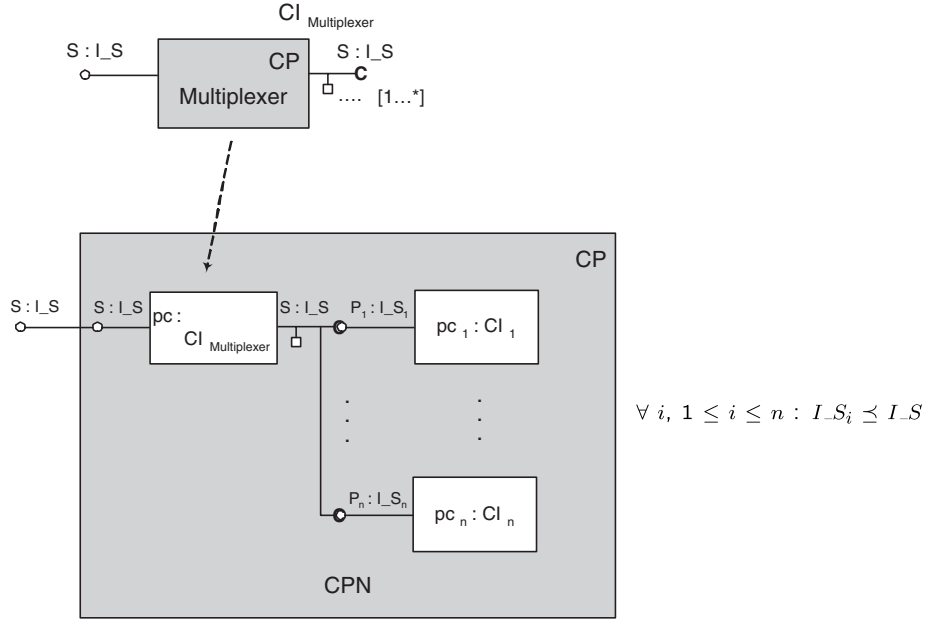


Figure 4.30: Composite UCM-Component with integrated Multiplexer Component

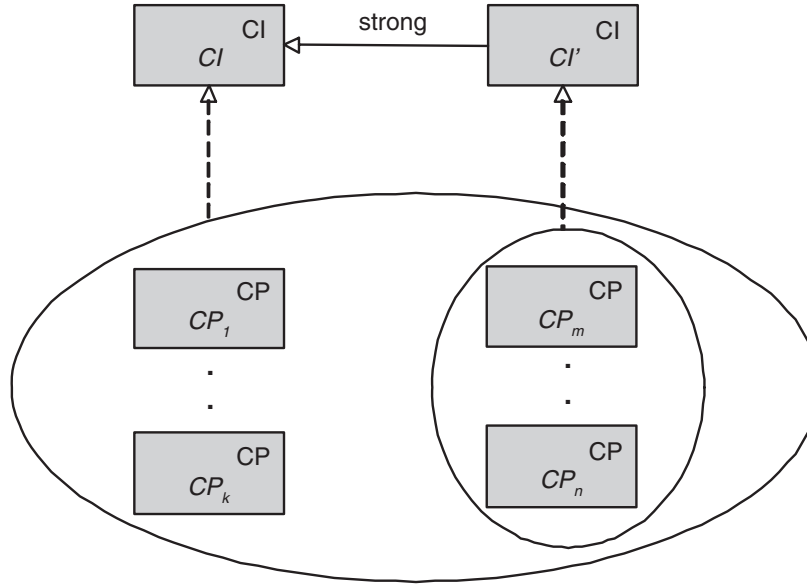
A multiplexer component can e.g. be used, if an instance of a composite UCM-component is asked to perform a task and in turn some or all of its parts also have to perform the same task. Examples are initialization tasks, changes to the appearance of visible parts (resizing, repainting, moving, getting visible or invisible, ...), data storing etc. Without using multiplexer, all provided services of all parts providing this task have to be exported and the task called on all these provided services by the client of the composite UCM-component instance.

4.8 Compatibility / Substitutability

4.8.1 Polymorphic Component Instances

Let pc be a part of a composite UCM-component CC and let pc be typed by the component interface CI . If neither CI nor pc is bound to a special component implementation in CC , we can distinguish three categories of components which can be used to instantiate pc .

1. **Arbitrary components implementing CI :** Such components can be used to instantiate pc by nature.
2. **Components implementing CI' with $Ctype(CI') \preceq Ctype(CI)$:** For this kind of components we will show that they can be used instead of a component implementing CI . Such components can be regarded as if they were not only implementing CI' but also CI . CI' can be regarded as extending CI . This view is shown in Figure 4.31. This figure shows k components CP_1, \dots, CP_k each implementing the component interface CI and the components CP_m, \dots, CP_n , $k < m \leq n$, each implementing CI' . The components implementing CI' are depicted as a subset of those implementing CI . The dotted arrow denotes an implementation relationship between components and their component interface.

Figure 4.31: Strong Subtyping between CI and CI'

3. **Components implementing CI' with $Ctype(CI') \preceq Ctype(CI)$:** For this kind of components we show that there is a simple means to wrap them into composite UCM-components implementing CI which can then be used to instantiate pc . This situation is depicted in Figure 4.32.

In this figure $CP'_1 - CP'_n$ are components implementing CI' . $CP_1^{CI} - CP_n^{CI}$ are the composite UCM-components wrapping $CP'_1 - CP'_n$ thereby implementing CI . $CP_1 - CP_k$ are other components implementing the component interface CI .

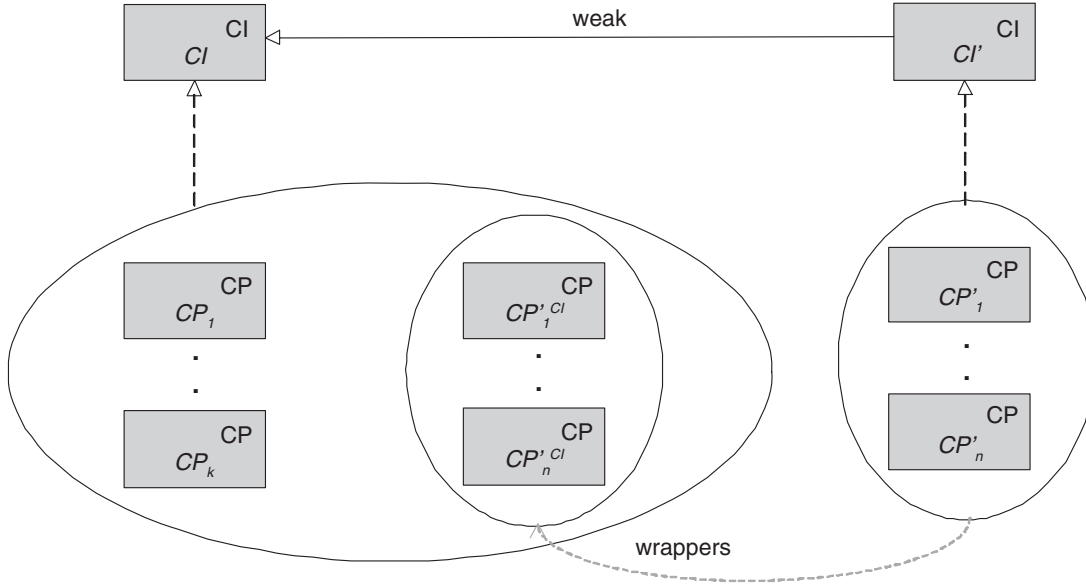


Figure 4.32: Weak Subtyping between CI and CI'

Category 2: Components implementing CI' with $\text{CItype}(\text{CI}') \preceq \text{CItype}(\text{CI})$:

Theorem 4.8.1 (Polymorphic Component Instances) *Let pc be a part of a composite UCM-component CC and let pc be typed by the component interface CI that is, $pc : CI$. Let neither pc nor CI be bound to a component implementation in CC . Let C' be a component implementing CI' with $\text{CItype}(\text{CI}') \preceq \text{CItype}(\text{CI})$. Then C' can be used to instantiate pc without invalidating CC .*

Proof: If we want to ensure that CC is not invalidated, all services of CI involved in a connection via pc must also be available in CI' . The corresponding services in CI' must still allow one to establish the connections declared in CC for pc . In addition, all services of CI involved in a link via pc to services of the component interface of CC must also be available in CI' . The corresponding services in CI' must still allow one to establish the links declared in CC for pc . For plugs, similar conditions have to hold. If all connections to required services of pc defined in CC and all links declared for required services of pc are established, there must not remain any open mandatory requirement for pc . The constraints declared in CI' must not be violated by the connections to required services of pc defined in CC . These conditions are described in more detail below. The component interface of CC is denoted by CI_{CC} .

1. The names of all services of CI which are referred to via pc in CC are also available in CI' .
2. If a provided service P of CI was connected via pc to a required service R of another part in CC typed by CI_{part} , then the provided service P of CI' can be connected instead.

(That is, if $pc : CI \wedge pc_{part} : CI_{part} \wedge pc_{part}.R \leq pc.P$ holds, then $pc_{part}.R \leq pc.P$ remains valid, if the runtime type of pc is CI' instead of CI .²¹)

3. If a required service R of CI was connected via pc to a provided service P of another part in CC typed by CI_{part} , then the required service R of CI' can be connected instead.

(That is, if $pc : CI \wedge pc_{part} : CI_{part} \wedge pc.R \leq pc_{part}.P$ holds, then $pc.R \leq pc_{part}.P$ remains valid, if the runtime type of pc is CI' instead of CI .)

4. If a provided service P of CI was linked via pc to a provided service P_{CC} of the component interface CI_{CC} of CC , then the provided service P of CI' can be linked instead.

(That is, if $pc : CI \wedge CI_{CC}.P_{CC} \leq pc.P$ holds, then $CI_{CC}.P_{CC} \leq pc.P$ remains valid, if the runtime type of pc is CI' instead of CI .²²)

5. If a required service R of CI was linked via pc to a required service R_{CC} of the component interface CI_{CC} of CC , then the required service R of CI' can be linked instead.

(That is, if $pc : CI \wedge CI_{CC}.R_{CC} \leq pc.R$ holds, then $CI_{CC}.R_{CC} \leq pc.R$ remains valid, if the runtime type of pc is CI' instead of CI .)

6. The names of all plugs of CI which are referred to via pc in CC are also available in CI' .

7. If a plug Pl of CI was connected via pc to a complementary plug Pl' of another part in CC typed by CI_{part} , then the plug Pl of CI' can be connected to Pl' instead.

(That is, if $pc : CI \wedge pc_{part} : CI_{part} \wedge pc.Pl \leq pc_{part}.Pl'$ holds, then $pc.Pl \leq pc_{part}.Pl'$ remains valid, if the runtime type of pc is CI' instead of CI .)

8. If a plug Pl of CI was linked via pc to a plug Pl_{CC} of the component interface CI_{CC} of CC , then the plug Pl of CI' can be linked to Pl_{CC} instead.

(That is, if $pc : CI \wedge CI_{CC}.Pl_{CC} \leq pc.Pl$ holds, then $CI_{CC}.Pl_{CC} \leq pc.Pl$ remains valid, if the runtime type of pc is CI' instead of CI .)

9. If all connections to required services of pc defined in CC and all links declared for required services of pc are established, there must not remain any open mandatory requirement for pc .

10. The constraints declared in CI' must not be violated by the connections to required services of pc defined in CC .

²¹Remember that ' \leq ' denotes a connection between a provided and a required service or between complementary plugs.

²²Remember that ' \leq ' denotes a link between services of the same kind (export) or between plugs.

Ad 1, Ad 6: As CI' is a strong subtype of CI , every service and plug declared in CI has a counterpart in CI' which is named identically (see subtype definitions 4.3.23 and 4.3.18).

Ad 2: As P of CI is connected via pc to a required service R of another part of CC typed by CI_{part} , $CI.P$ ²³ and $CI_{part}.R$ have to be matching services. We have to show that then also $CI'.P$ and $CI_{part}.R$ are matching services²⁴. Then we know from theorem 4.4.6 that $CI'.P$ can also be connected to $CI_{part}.R$.

$CI.P$ matches $CI_{part}.R \iff \text{Sltype}(CI, P) \preceq \text{Sltype}(CI_{part}, R)$ (term definition 4.4.5)

$\text{Cltype}(CI') \preceq \text{Cltype}(CI) \implies \text{Sltype}(CI', P) \preceq \text{Sltype}(CI, P)$
(see subtype definition 4.3.23)

$\implies \text{Sltype}(CI', P) \preceq \text{Sltype}(CI_{part}, R)$ (theorem 4.3.12)

$\iff CI'.P$ matches $CI_{part}.R$ (term definition 4.4.5)

Ad 3: For $CI'.R$ we have to show that:

1. $CI'.R$ and $CI_{part}.P$ are matching services,
2. the connection point type $\text{CPtrtype}(CI', R)$ contains the connect-method used to establish the connection between $CI_{part}.P$ and $CI.R$ via pc in CC ,
(This is due to the fact that for every connection, the connect-method used to establish the connection is stored in CC or CI_{CC} . These methods are called when instantiating CC . Therefore $CI'.R$ has also to provide the connect-method stored in CC or CI_{CC} for the connection between $CI_{part}.P$ and $CI.R$ via pc .)
3. the conditions for the limits on the number of connections to $CI'.R$ via pc are satisfied, if they are satisfied for $CI.R$.

ad 1 As the required service R of CI was connected via pc to a provided service P of another part in CC typed by CI_{part} , $CI.R$ and $CI_{part}.P$ are matching services.

$CI_{part}.P$ matches $CI.R \iff \text{Sltype}(CI_{part}, P) \preceq \text{Sltype}(CI, R)$

$\text{Cltype}(CI') \preceq \text{Cltype}(CI) \implies \text{RStype}(CI', R) \preceq \text{RStype}(CI, R)$
(subtype definition 4.3.23)

$\implies \text{Sltype}(CI', R) \preceq \text{Sltype}(CI, R)$ (type definition 4.3.5 and
subtype definition 4.3.15)

$\implies \text{Sltype}(CI_{part}, P) \preceq \text{Sltype}(CI', R)$ (theorem 4.3.12)

$\iff CI_{part}.P$ matches $CI'.R$ (term definition 4.4.5).

²³To distinguish the provided service P of CI from P of CI' , we qualify the service names with the names of the component interfaces they belong to.

²⁴We also say that P matches R .

ad 2 Let $m : MT^m \in \text{CType}(CI, R)$ be the connect-method used to establish the connection between $CI_{part}.P$ and $CI.R$ via pc in CC .

$$\text{CType}(CI') \preceq \text{CType}(CI) \implies \text{RType}(CI', R) \preceq \text{RType}(CI, R)$$

(subtype definition 4.3.23)

$$\implies \text{CType}(CI', R) \preceq \text{CType}(CI, R) \text{ (type definition 4.3.5 and subtype definition 4.3.15)}$$

$$\text{CType}(CI', R) \preceq \text{CType}(CI, R) \iff \text{CType}(CI', R) \supseteq \text{CType}(CI, R).$$

$$\implies m : MT^m \in \text{CType}(CI', R)$$

Thus, m can also be used to establish the connection between $CI_{part}.P$ and $CI'.R$ via pc .

ad 3 To be able to distinguish lower and upper limits on the number of connections for R of CI and R of CI' , we refer to \min_R and \max_R as $\min_{CI.R}$, $\max_{CI.R}$ and $\min_{CI'.R}$, $\max_{CI'.R}$. We assume that the conditions are satisfied for $CI.R$ via pc in CC .

Let $\#conn$ be the number of connections defined in CC to $CI.R$ via pc .

$$\implies \min_{CI.R} \leq \#conn \leq \max_{CI.R}.$$

$$\text{CType}(CI') \preceq \text{CType}(CI) \implies \text{Cardtype}(CI', R) \preceq \text{Cardtype}(CI, R)$$

(see subtype definitions 4.3.23 and 4.3.15)

$$\implies \min_{CI'.R} \leq \min_{CI.R} \wedge \max_{CI.R} \leq \max_{CI'.R} \text{ (subtype definition 4.3.14)}$$

$$\implies \min_{CI'.R} \leq \#conn \leq \max_{CI'.R}.$$

Ad 4: P of CI can only be linked via pc to a provided service P_{CC} of the component interface CI_{CC} of CC , if $\text{PType}(CI_{CC}, P_{CC}) \succeq \text{PType}(CI, P)$ (*).

We have to show that then also $\text{PType}(CI_{CC}, P_{CC}) \succeq \text{PType}(CI', P)$. Then we know from theorem 4.4.10 that P of CI' can also be linked to P_{CC} .

$$\text{CType}(CI') \preceq \text{CType}(CI) \implies \text{PType}(CI', P) \preceq \text{PType}(CI, P) \quad (**)$$

(see subtype definition 4.3.23)

$$(*), (**) \implies \text{PType}(CI', P) \preceq \text{PType}(CI_{CC}, P_{CC}) \text{ (theorem 4.3.12)}$$

$$\iff P \text{ of } CI' \text{ can be linked to } P_{CC} \text{ (theorem 4.4.10)}$$

Ad 5: For this proof, similar steps are necessary as in the previous proofs. We need the transitivity of the subtype relation between required services (theorem 4.3.16), the definition of a strong subtype relation between component interfaces (subtype definition 4.3.23) and theorem 4.4.12.

Ad 7: It remains to show that, if $CI.Pl$ and $CI_{part}.Pl'$ are complementary plugs, $CI'.Pl$ and $CI_{part}.Pl'$ are also complementary plugs. Then the proposition follows directly from theorem 4.4.8.

$$CI.Pl \text{ complementary to } CI_{part}.Pl' \iff \text{(term definition 4.4.7)}$$

1. $\exists f : \text{Provided}(CI, Pl) \longrightarrow \text{Required}(CI_{part}, Pl'), f$ is bijective with
 $\forall P \in \text{Provided}(CI, Pl) : \text{PStype}(CI, P) \preceq \text{SItype}(CI_{part}, f(P))$
2. $\exists g : \text{Provided}(CI_{part}, Pl') \longrightarrow \text{Required}(CI, Pl), g$ is bijective with
 $\forall P' \in \text{Provided}(CI_{part}, Pl') : \text{PStype}(CI_{part}, P') \preceq \text{SItype}(CI, g(P'))$

We have to show: $CI'.Pl$ is complementary to $CI_{part}.Pl'$, that is S 1 and S 2 hold.

S 1 $\exists f' : \text{Provided}(CI', Pl) \longrightarrow \text{Required}(CI_{part}, Pl'), f'$ is bijective with
 $\forall P \in \text{Provided}(CI', Pl) : \text{PStype}(CI', P) \preceq \text{SItype}(CI_{part}, f'(P))$

S 2 $\exists g' : \text{Provided}(CI_{part}, Pl') \longrightarrow \text{Required}(CI', Pl), g'$ is bijective with
 $\forall P' \in \text{Provided}(CI_{part}, Pl') : \text{PStype}(CI_{part}, P') \preceq \text{SItype}(CI', g'(P'))$

$\text{CIttype}(CI') \preceq \text{CIttype}(CI) \implies \text{PLtype}(CI', Pl) \preceq \text{PLtype}(CI, Pl)$
 (see subtype definition 4.3.23)

(with subtype definitions 4.3.18 and 4.3.17) \implies

1. $\text{Provided}(CI, Pl) = \text{Provided}(CI', Pl)$ (*)
2. $\text{Required}(CI, Pl) = \text{Required}(CI', Pl)$
3. $\forall P \in \text{Provided}(CI, Pl) : \text{PStype}(CI', P) \preceq \text{PStype}(CI, P)$ (**)
4. $\forall R \in \text{Required}(CI, Pl) : \text{RStype}(CI', R) \preceq \text{RStype}(CI, R)$

$CI.Pl$ complementary to $CI_{part}.Pl' \implies \text{PStype}(CI, P) \preceq \text{SItype}(CI_{part}, f(P))$ (***)

(**), (***) $\implies \text{PStype}(CI', P) \preceq \text{PStype}(CI, P) \preceq \text{SItype}(CI_{part}, f(P))$

$\implies \text{PStype}(CI', P) \preceq \text{SItype}(CI_{part}, f(P))$ (theorem 4.3.12)

(Theorem 4.3.12 can be used as $\text{PStype}(CI', P)$, $\text{PStype}(CI, P)$ and $\text{SItype}(CI_{part}, f(P))$ are method based interface types, see type definitions 4.3.3 and 4.3.4.)

(*) $\implies f'$ can be set to $f \implies \text{PStype}(CI', P) \preceq \text{SItype}(CI_{part}, f'(P))$

Thus **S 1** holds. Similarly, **S 2** can be shown.

Ad 8: The proof follows from the definition of a strong subtype relation between component interfaces (subtype definition 4.3.23), the transitivity of the subtype relation between plugs (theorem 4.3.19), and the theorems concerning linking between plugs (4.4.16 and 4.4.17).

Ad 9: The proof can be done similar to Ad 4 in the proof of theorem 4.4.12 based on the subtype relations for the cardinality types for required services.

Ad 10: We have to show that, if pc holds an instance of type CI' instead of type CI , the constraints declared by CI' are not violated by the connections to required services of pc defined in CC which only respect the constraints declared for CI . That is, if R'_1 and R'_2 are two required services of CI' which are also declared in CI

and which belong to a constraint C' of CI' then the service providers connected to them via pc are distinct from each other.

The proposition follows directly from item 6 of the subtype definition for component interfaces 4.3.22 which enforces that then $g^{-1}(R'_1)$ and $g^{-1}(R'_2)$ both belong to a common constraint in CI , too. $g: \text{Required}(CI) \rightarrow \text{Required}(CI')$ is the mapping from subtype definition 4.3.22 which defines counterparts in CI and CI' . Since $\text{Ctype}(CI') \preceq \text{Ctype}(CI)$, g is the identity mapping.

If CI' declares a constraint which contains a required service which is not declared in CI , this required service can not lead to a violation, because CC only declares connections to required services of CI .

□

Category 3: Components implementing CI' with $\text{Ctype}(CI') \preceq \text{Ctype}(CI)$: For this kind of components we show that there is a simple means to wrap them into components implementing CI which can then be used to instantiate pc .

If CI' is only a *weak* subtype of CI that is, $\text{Ctype}(CI') \preceq \text{Ctype}(CI)$, service and plug names of equivalent entities may differ. Therefore, the service and plug names of CI have to be mapped to the names of the equivalent services and plugs of CI' . This can be done by creating a composite UCM-component $CP_Wrapper_CI'$ implementing CI and having a sole part $pc_Original$ of type CI' . All services and plugs of CI' equivalent to services and plugs of CI are exported and linked to the equivalent entities of CI . If f denotes the mapping for the provided services, g the mapping for the required services and h the mapping for the plugs used in subtype definition 4.3.22, $CP_Wrapper_CI'$ looks as follows:

```

Component CP_Wrapper_CI' implements CI {
  GeneralDescriptions
    type          = composite

  Parts
    P_Original : CI'

  InternalConnections

  Exports
    ProvidedServices
      CI.pl <-- P_Original.f(pl)
      ...
      CI.pn <-- P_Original.f(pn)

```

```

RequiredServices
  CI.r1 <-- P_Original.g(r1)
  ...
  CI.rm <-- P_Original.g(rm)

Plugs
  CI.pl1 <-- P_Original.h(pl1)
  ...
  CI.plk <-- P_Original.h(plk)

ImplementationBinding
  ...
}

```

where $f(pi)$, $g(ri)$ and $h(pli)$ represent the names of the corresponding entities in CI' as e.g. pi' , ri' , pli' . Export definitions for plugs have eventually to be refined according to the rules discussed in Section 4.2.3.2.

Although a component CP' implementing CI' can not directly be used to instantiate pc of CC , $CP_Wrapper_CI'$ can be used instead. If one wants to ensure that CP' is used for instantiation of $P_Original$, the 'ImplementationBinding' section of $CP_Wrapper_CI'$ has to contain the binding $CI' <<< CP'$. Otherwise every other component implementing CI' can also be used for instantiation.

$CP_Wrapper_CI'$ does not only map names of CI to names of CI' , it automatically also maps subtypes to supertypes and thus realizes some kind of type cast. This is due to the fact that the services and plugs of CI may be supertypes of the equivalent services and plugs of CI' which follows from $CItype(CI') \preceq CItype(CI)$. By linking the services and plugs of CI' to equivalent services and plugs of CI , an implicit upcast is realized. In the life-cycle of a component, e.g. service interface types will be specialized and the corresponding service names will be changed to express the difference to a former release. Thus this kind of subtyping is natural. In addition, if a component interface CI' is a weak subtype of a component interface CI , the wrapper described above can be created simply by using the export-mechanism. If the mapping between the names of equivalent services and plugs of CI and CI' is unambiguous, then this wrapper can even be created automatically by a tool e.g. at deployment time.

Other Wrappers A wrapper-mechanism can also be used for other purposes like adapting an existing, incompatible service interface type to the needed service interface type. But in this case, creating a suitable wrapper needs the aid of a person who selects or implements a suitable adapter converting the incompatible service interface type to the needed service interface type and who combines the adapter and the existing component properly to build a wrapper implementing the needed interface. Such a process is referred to as *component adaptation*. We do not consider such changes in the context of subtype relations.

4.8.2 Replacing Components because of Upgrades or Change of Vendors

To remain flexible, companies want to be able to use new versions of already installed components or to switch to components of other vendors which are more tailored to the needs of the company as the old ones.

If a company does not want to maintain the formerly used components further more and wants additionally ensure that all applications and components built using the former components can still be used without changes made to their code, if the former components are replaced by new ones, the new components have to obey certain rules.

Thus, in this section we shall examine when an UCM-component can be replaced by another one without invalidating any existing composite UCM-component which internally refers to the UCM-component to be replaced.

For this purpose we have to distinguish two kinds of composite UCM-components:

1. composite UCM-components which internally only refer to the component interface of the component to be replaced,
2. composite UCM-components which in addition refer to the component implementation of the UCM-component to be replaced.

Theorem 4.8.2 (Compatible Components) *Let CP with component interface CI_{CP} and CP' with component interface $CI_{CP'}$ be two UCM-components with $CType(CI_{CP'}) \preceq CType(CI_{CP})$. Then*

1. *CP can be replaced by CP' without invalidating any composite UCM-component which internally only refers to CI_{CP} , but not to the component implementation CP itself.*
2. *CP can be replaced by CP' without invalidating any existing UCM-component, if CP and CP' are named equally or if CP' is registered as a valid substitution for CP .*

Proof:

Ad 1 If CP is replaced by CP' , CP is no longer available. Instead of CP , CP' has to be used to instantiate all parts of composite UCM-components which were formerly instantiated by CP . Theorem 4.8.1 guarantees that for all composite UCM-components CC which contain parts pc typed by CI_{CP} and which do not refer to CP in their ImplementationBinding section, CP' can be used for instantiation instead of CP without invalidating CC .

Ad 2 Let CC be a composite UCM-component directly referring to CP in its implementation binding section that is, $CI_{CP} <<< CP$ or $pc <<< CP$ for some pc in CC typed by CI_{CP} . If CP and CP' are named equally, the implementation binding statements need not be changed.

The situation is more complicated, if the name of CP' is distinct from CP . If the runtime system tries to instantiate CC , it has to instantiate all parts pc belonging to CC which are typed by CI_{CP} including those for which CP was declared in the implementation binding section of CC to be used for instantiation. As CP can no longer be found in the implementation registry (Section 4.5), since it was replaced by CP' the instantiation process can only be terminated successfully, if CP' is registered as a valid substitution for CP in the substitution registry. For more details on this process please refer to Sections 4.5 and 4.6.

□

Even UCM-components CP' with $\text{CType}(CI_{CP'}) \preceq \text{CType}(CI_{CP})$ can be used for substitution. But instead of replacing CP by CP' , CP is replaced by the wrapper component corresponding to CP' as described in Section 4.8.1 (category 3).

4.9 Realisation of Composite UCM-Components

Before showing that and how industrial component models can be integrated into our model, we show how composite UCM-components can be realized and how component instances, especially instances of composite UCM-components, can be referred to by a tool or in the context of a programming language. The discussions of this section give some insight on what remains to show for the integration of components belonging to industrial component models. We especially introduce the basics which are needed to show that instances of composite UCM-components can be built from the implementation description of a composite UCM-component for each component model under consideration.

Instead of referring to pure component instances represented by their service access interface, our runtime system uses *component info objects* which internally refer to the service access interface of the component instance they represent and which provide additional information as e.g. on the component interface, the component instance implements. In addition to the possibility to access the functionality of a component instance via its service access interface, a component info object allows one to query type meta information about the corresponding component interface. On demand, the runtime system only returns a reference to the object implementing the service access interface instead of a reference to the component info object.

A component info object consists of a *component interface info object* and a *component implementation info object*.

The component interface info object holds the name of the component interface implemented by the component instance (field `c_InterfaceName` in Figure 4.33) and it holds a reference to a *component interface object* (field `c_InterfaceSpec` in Figure 4.33). The component interface object is built based on the information available in the component interface specification. It can be queried for information on the services a component provides or requires, their service interface types, the plugs defined, the default connect-methods declared for required services, the upper limit on the number of

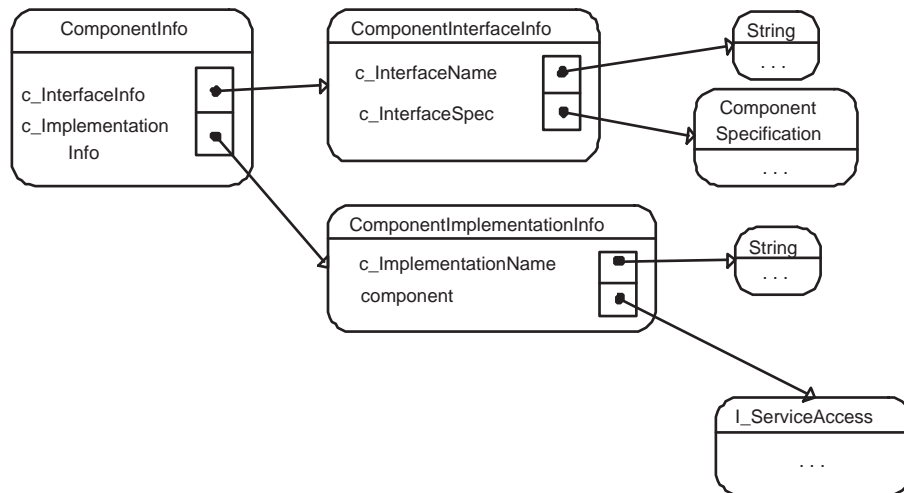


Figure 4.33: Component Info Object

connections to a required service, whether a required service is optional or mandatory and so forth. Thus, this object serves to provide type meta information on the implemented component interface. Some of the methods which have to be implemented by component interface objects are listed below. The programming language used is Java.

```

public interface ComponentSpecification {

    // Returns the names of all services provided by the component.
    public String[] getProvidedServiceNames ();

    // Returns the names of all services required by the component.
    public String[] getRequiredServiceNames ();

    // Returns the names of all mandatory required services.
    public String[] getMandatoryConnections ();

    // Returns the upper limit on the number of connections to the
    // required service 'serviceName'.
    public int getMaxNumberOfConnections (String serviceName);

    public int getMinNumberOfConnections (String serviceName);

    public boolean mandatoryConnection (String rServiceName);

    public String getServiceTypeAsString (String serviceName,
                                           boolean providedService);

    public Class getServiceType(String serviceName, boolean providedService);
  
```



```

// Returns whether a default connect-method is specified for the
// required service 'rServiceName'.
public boolean existsMethod(String rServiceName);

// Returns the name of the default connect- or disconnect-method depending on
// the value of the parameter 'connectMethod'. 'True' signals to look for the
// connect-method, 'false' signals to look for the disconnect-method.
public String getMethodAsString (String rServiceName, boolean connectMethod);

public int countMethodParameters(String rServiceName, boolean connectMethod);

// Returns the names of all parameter types for the default connect-method
// corresponding to the required service 'rServiceName'.
public String[] getConnectMethodParamsAsString (String rServiceName);

...
}

```

The component implementation info object holds the name of the component implementation and it holds a reference to a *component implementation object* (field component in Figure 4.33). The component implementation object implements `I_ServiceAccess` that is, it represents the service access interface of the component instance. Via this service access interface all services and connection points of the component instance can be accessed. The component instance is built based on the information available in the component implementation file.

If the component implementation refers to an atomic UCM-component, the type of the component implementation object depends on the industrial component model used for atomic UCM-components and on whether the component instance already implements `I_ServiceAccess` or not. In case of JavaBeans, this object is an instance of the `JavaBean`-class, if this class implements `I_ServiceAccess`. Otherwise, it is an instance of a special wrapper class only implementing `I_ServiceAccess` and internally referring to an instance of the `JavaBean`-class. For COM components, the component implementation object is an instance of the class `AtomicComp` (see Section 4.10.2.1). For more details on these wrapper classes please refer to the explanations on how industrial component models are integrated into our model in Section 4.10.

The implementation of the class used as type for the component implementation object in case of composite UCM-components only differs slightly depending on the industrial component model used for atomic UCM-components. Differences appear when interconnections between services of two parts have to be established. For this purpose, the kind of the type metadata and the methods available to explore connection point types unknown at compile time and to invoke methods on them have to be used. This varies from component model to component model. For more details on this subject see the subsections of Section 2.2 concerning type metadata and Section 4.10.

Apart from these differences, composite UCM-components are realized by the same wrapper class. This class implements `I_ServiceAccess` and holds information on

1. the export declarations made in the component implementation of the composite UCM-component and
2. a list of objects representing its parts. For every part its name is stored as well as a reference to a component info object representing the corresponding component instance.

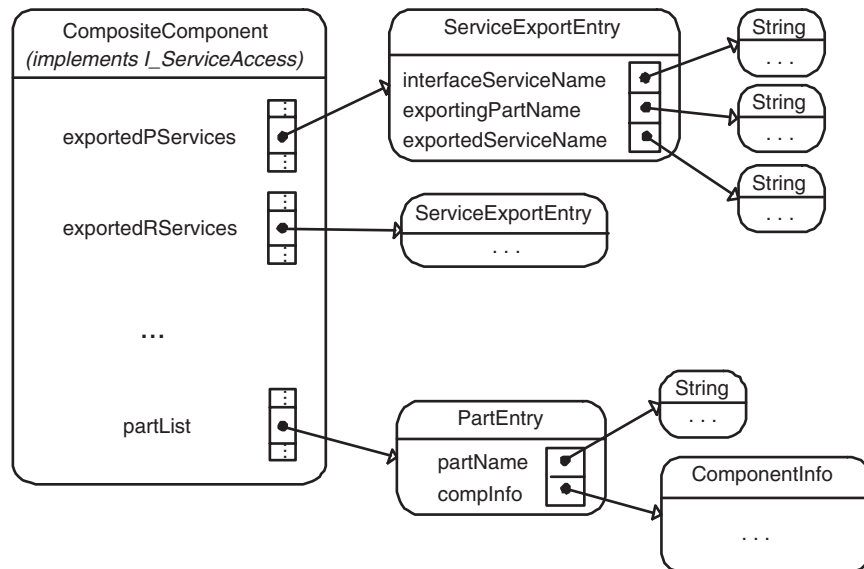


Figure 4.34: Instance of a Composite UCM-Component

As already described above, each component info object internally refers to a component implementation object implementing `I_ServiceAccess`. These component implementation objects are instances of the wrapper class for composite UCM-components or instances of atomic UCM-components depending on the kind of the component implementation (composite, atomic) selected for this part.

The service access interface of the wrapper class for composite UCM-components delegates a call to `getServiceReference` or `getConnectionPointObject` to the service access interface of the wrapper or maybe bean-class instance stored in the component info object of the part exporting the requested service. The delegation process terminates on a wrapper or maybe bean-class instance of an atomic UCM-component. See also Figure 4.28.

The interface of the class used to instantiate component info objects is shown below. It is written in Java. `ComponentInterfaceInfo` is the class used to instantiate component interface info objects and `ComponentImplementationInfo` is the class used to instantiate component implementation info objects. `ComponentImplInfos` is a class representing the parsed contents of the component implementation file. Depending on whether the component implementation specifies an atomic or composite UCM-component, instances of this class can be queried for the industrial component

model used, for the identification of the component to be instantiated (e.g. CLSID), for the parts declared, for the export definitions and so forth. The declaration of the interface `ComponentSpecification` was already shown above.

```
public class ComponentInfo {
    private ComponentInterfaceInfo      c_InterfaceInfo;
    private ComponentImplementationInfo c_ImplementationInfo;

    public ComponentInfo(ComponentImplInfos cii) throws ComponentException {
        ...
    }

    // Returns the name of the component implementation
    public String getComponentName() {
        ...
    }

    // Returns a reference to the component implementation object
    public I_ServiceAccess getComponent(){
        ...
    }

    // Sets the reference to the component implementation object
    public void setComponent(I_ServiceAccess c){
        ...
    }

    // Returns a reference to the component interface object
    public ComponentSpecification getComponentSpec () {
        ...
    }

    // Sets the reference to the component interface object
    public void setComponentSpec (ComponentSpecification cpSpec) {
        ...
    }
}
```

The following description is based on Figure 4.34.

The export information contained in an instance of a composite UCM-component is used to delegate a call on its service access interface to the service access interface of the part exporting the service. Thus, if an instance of a composite UCM-component is queried for a service object bound to one of its provided services, the corresponding `ServiceExportEntry` is looked up from the list of exported provided services (`exportedPServices` in Figure 4.34). This entry contains the name of the part exporting the service (`exportingPartName`) as well as the name of the part's own provided service (`exportedServiceName`) implementing the service of the composite UCM-component. Having the name of the part, the corresponding `PartEntry` can be looked up in the list `partList`. This entry holds a reference to the component info

object (`compInfo`) representing the component instance corresponding to the part. Using `getComponent()`, the component info object can be queried for a reference to the component implementation object which is of type `I_ServiceAccess`. On this object, `getServiceReference` is called with the service name of the part used to implement the service of the composite UCM-component. The service object returned is in turn returned by `getServiceReference` called on the instance of the composite UCM-component. The delegation process terminates at a component implementation object representing an atomic UCM-component (see Section 4.6). A similar process is used to query an instance of a composite UCM-component for a connection point object corresponding to one of its required services. This way, the links between the services of a composite UCM-component with interface CI and the services of its parts, e.g. $CI.P \leftarrow part.P_1$, are realized.

When creating an instance of a composite UCM-component the information on how its parts are interconnected have to be evaluated. We focus on interconnections on service level as interconnections on plugs can be reduced to interconnections on service level. A full-fledged connect entry in the component implementation of a composite UCM-component between a required service R of a part pc and a provided service P of a part pc' looks as follows (see Section 4.2.3.1):

$$pc.R \Leftarrow (methodName(type-par_1 \text{ value-}par_1, \dots, type-par_m \text{ value-}par_m)) pc'.P$$

The corresponding information can be modelled by the following class:

```
public class InternalConnection {

    public String serviceRequester;
    public String rServiceName;

    public String serviceProvider;
    public String pServiceName;

    public Vector conMethParams;
    public String conMethName;

    ...
}
```

`serviceRequester` holds the name of the part with the required service and `rServiceName` the name of the required service. Similarly, `serviceProvider` holds the name of the part with the provided service and `pServiceName` the name of the provided service. `conMethName` holds the name of the connect-method and `conMethParams` holds for each parameter one string-value with the parameter type and one with the value of the actual parameter. Thus, having the name of a part, the corresponding part entry can be looked up and in turn the component implementation object of type `I_ServiceAccess` retrieved as already shown in the context of export declarations. In the following, we refer to the component implementation object of the part having the required service as *obj_serviceRequester* and the component implementation

object of the part having the provided service as *obj_serviceProvider*. The service object bound to *pServiceName* can be retrieved by

```
obj_serviceProvider.getServiceReference(pServiceName)
```

and the connection point object belonging to *rServiceName* by

```
obj_serviceRequester.getConnectionPointObject(rServiceName).
```

In the following, the service object is referred to by *servObj* and the connection point object by *conObj*. To establish the connection, the connect-method must be invoked on *conObj* taking as actual parameter *servObj* for the formal parameter declared to hold the needed reference. To be able to do so, we must use the reflection services available for the industrial component model under consideration to retrieve a method object from its string representation in *InternalConnections* and to create the actual parameters other than *servObj* from their string representations. With these actual parameters, the method is then called by the invoke-mechanisms available through reflection. The position of the parameter to be used to pass *servObj* in the parameter list can be determined from *conMethParams* by looking for the parameter with the argument value “-”.

4.10 Integration of Industrial Component Models

Components of existing component models can only be integrated, if they provide an interface and implementation specification as defined in Figures 4.11 on page 111 and 4.15 on page 122. From these specifications component info objects as presented in Section 4.9 will be created representing a component instance.

The contained component implementation object implementing the service access interface is either an instance of a component of the underlying component model or an instance of a special wrapper class implementing the service access interface. This special wrapper class internally refers to an instance of a component of the underlying component model. The wrapper class is needed to enable the integration of components which are still unaware of our component model and will thus not implement the service access interface.

The component interface object contained in the component info object is queried for the services and plugs declared as well as the types of the service interfaces. Amongst others, these types are used to check, whether two types are related by a subtype relationship.

Therefore, to show that an existing industrial component model can be integrated into our model we have to answer the following questions:

1. What is a suitable declaration of *I_ServiceAccess*?

As the methods *getServiceReference* and *getConnectionPointObject* must return references to service objects of an arbitrary service interface type and connection point objects of an arbitrary connection point type, the return type of these methods must cover all these types.

2. How does our component interface specification look like for this industrial component model? Especially: how are service interface types denoted?

Based on the denotation of the service interface types, we must be able to create a type representation which can be used to check service interface types for a subtype relationship. This check is needed for our subtype relations declared for provided and required services, plugs and component interfaces. The subtype relation is essentially used by the rules for substitutability, checks for the correctness of a composition, and by tools supporting features like determining a set of suitable service providers (see Section 5.1).

In this context we show, how a check for a subtype relationship can be implemented.

3. How does a component implementation for an atomic UCM-component look like? Especially: how is a component of the industrial component model referred to and how can an instance be built from this information?

The information must allow one to look up the component needed and to instantiate it. For this purpose we need the mechanisms for component lookup and instantiation described for the different industrial component models in item *Component lookup, instantiation and access* of their description in Section 2.2.

4. Can a composite UCM-component be built from its component implementation specification? Especially: Can a connection be established based on the information on the connect-method in the component implementation or on the default connect-method in the component interface specification?

From Section 4.9 we know already that links/exports can be realized by a certain implementation of the service access interface of a composite UCM-component. In addition, we know, how the connection point object (*conObj*) and service object (*servObj*) corresponding to a connect-entry in the section 'InternalConnections' can be obtained. From Section 4.9 we also know which steps are necessary to end up in a connection using the connect-method specification.

Following these steps, it remains to show that it is possible

- to determine a method object representing the connect-method which can then be used to call this method. It must be possible to determine this method object only based on its name and parameter types given as strings.
- to generate arguments for the method call based on the values given as strings. (For the argument with value "-", *servObj* has to be used as actual parameter.)
- to execute the connect-method on the connection point object.

For this purpose we need the information on the type metadata and the reflection services described for every component model in their subsections titled *Type Metadata* of Section 2.2.

To be capable of using the full power of our model, we have to show additionally that

1. an implementation of `I.ServiceAccess` can be provided by a special wrapper for atomic UCM-components, if the component of the industrial component model does not implement this interface by itself. That is, the component was not yet designed with respect to our component model, but can nevertheless be integrated by only providing a suitable component interface specification and a component implementation specification.
2. existing approaches for services or event connections can be integrated into our model.

(For this purpose we need the information on the component interface and composition techniques described for every component model in their subsections of Section 2.2 which are titled *Component Model* respectively *.NET Framework* and *Composition Techniques*.)

3. concepts like our services and plugs can be used even if related concepts are missing in the industrial component model under consideration. In addition, service and plug connections can be applied.

4.10.1 JavaBeans

4.10.1.1 Component Implementations and Component Interfaces Specifications

In this subsection we answer the first four questions from Section 4.10.

Ad 1 *What is a suitable declaration of `I.ServiceAccess`?*

For JavaBeans, `I.ServiceAccess` as declared in Section 4.1.1.1 on page 93 is suitable. References for all types of service interfaces and connection points can be obtained by calls to the methods of `I.ServiceAccess`, since `Object` covers all these types.

Ad 2 *How does our component interface specification look like for this industrial component model? Especially: how are service interface types denoted?*

Service interface types for JavaBeans are denoted by the names of their corresponding Java interfaces including package information. The fact that Java types are used to denote service interface types must be announced by setting the value for 'NamingConventions' in the component interface specification to 'JavaType'.

For examples of component interface specifications using JavaBeans refer e.g. to examples 4.1.23 and 4.2.1.

Subtype relations for service interface types can be determined as shown below, based on the names of their Java types being available as strings from the component interface specification. The Class objects corresponding to the service interface types can be obtained by calling `Class.forName` passing the string representations of the Java types as argument (see Section 2.2.1.4). A method similar to “isSubtype” is implemented in Sun’s BeanBox.

```
// Searches the chain of supertypes of 'a' by calling
// 'isSubtype' recursively. 'a' and 'b' represent interfaces.

static boolean isSubtype(Class a, Class b) {
    // We rely on the fact that for any given java class or
    // primitive type there is a unique Class object, so
    // we can use object equivalence in the comparisons.
    if (a == b) {
        return true;
    }
    if (a == null || b == null) {
        return false;
    }

    /* Determine all interfaces extended by interface 'a'. The order of
     * the interface objects in the array corresponds to the order of
     * the interface names in the extends clause of the declaration of
     * the interface represented by 'a'.
     */
    Class interfaces[] = a.getInterfaces();
    for (int i = 0; i < interfaces.length; i++) {
        if (isSubtype(interfaces[i], b)) {
            return true;
        }
    }
    return false;
}

...
// If "interfacel" and "interface2" are two names of Java interfaces
// and one wants to know whether interfacel is a subtype of interface2,
// one has to call
boolean b = isSubtype(Class.forName("interfacel"),
                      Class.forName("interface2"));
...
```

Ad 3 *How does a component implementation for an atomic UCM-component look like? Especially: how is a component of the industrial component model referred to and how can an instance be built from this information?*

In component implementations of atomic UCM-components the value for ‘ComponentModel’ is set to ‘JavaBeans’ and a JavaBean is referred to by the name of its JavaBean-class including package information.

An instance of the JavaBean can be created by a call to `java.beans.Beans.instantiate` which takes as input the string representation of the name of the JavaBean-class (see Section 2.2.1.1 item 4).

For examples of component implementations for atomic UCM-components using JavaBeans refer e.g. to example 4.2.1.

Ad 4 *Can a composite UCM-component be built from its component implementation specification? Especially: Can a connection be established based on the information on the connect-method in the component implementation or on the default connect-method in the component interface specification?*

For examples of component implementation specifications for composite UCM-components using JavaBeans refer e.g. to example 4.2.1. Example 4.2.2 on page 127 shows the declaration of a connect-method with additional parameters:

```
connect(IList -, String OrderList).
```

The data types used as parameter types in connect-method declarations are also denoted as Java types.

From Section 4.10 we know that it only remains to show that it is possible

- to determine a method object representing the connect-method which can then be used to call this method. It must be possible to determine this method object only based on its name and parameter types given as strings.
- to generate arguments for the method call based on the values given as strings. (For the argument with value "-", *servObj* has to be used as actual parameter.)
- to execute the connect-method on the connection point object.

Determine a method object: In the following, the name of the connect-method is referred to as `methodName`.

From the parameter types given as strings the corresponding Class objects representing their types can be obtained by a call to `Class.forName` (see Section 2.2.1.4). These Class objects can be stored in an array listing all parameter types (e.g. `Class[] parameterTypes`). The connection point type the connect-method belongs to can be retrieved from the connection point object *conObj* by calling `conObj.getClass()`; *conObj* can be retrieved as shown in Section 4.9. Having the Class object representing the connection point type one can call `getMethod` on this Class object to retrieve the method object representing the connect-method. The method object is of type `java.lang.reflect.Method` and is identified by the method name available as string and the parameter types available by an array of Class objects:

```
Method conMeth = conObj.getClass().getMethod(methodName, parameterTypes);
```

Before being able to execute the connect-method, we must be able to generate arguments for the method call based on the values given as strings.

Generate arguments: Since we restricted the argument types for connect-methods to primitive data types and the type of the corresponding required service, one can generate the arguments as follows. Arguments for primitive data types like `int`, `Integer`, `double`, `Double`, ... can be generated by calling the constructors of the corresponding Java classes resp. wrapper classes having a single parameter of type `String`. The string value available for the argument of the connect-method to be generated is passed as argument to the constructor like e.g. `Object value = new Integer(argAsString);`

An argument value of `"-"` for an argument of the connect-method indicates that the actual parameter is a reference to the object implementing the provided service. Therefore this argument can be set to `servObj` which can be retrieved as shown in Section 4.9.

Execute the connect-method: As already mentioned in Section 2.2.1.4, a method represented by an object of type `Method` can be executed by calling `Invoke` on this object. The first parameter of `Invoke` refers to the object the method is invoked from. In our case this is `conObj`. The second parameter refers to an array of objects representing the arguments of the method to be executed by `Invoke`. The following code snippet shows the fundamental steps for executing the connect-method.

```
Object result = null;
int numberParams = paramTypesAsString.length;
Object[] args = new Object[numberParams];

for (int j = 0; j < numberParams; j++) {
    args[j] = createArgument (paramValuesAsString[j],
                             paramTypesAsString[j], servObj);
}

result = conMeth.Invoke (conObj, args);
```

In the code above `paramValuesAsString` and `paramTypesAsString` contain the values and type names for the parameters of the connect-method as strings. `createArgument` represents a method generating an argument depending on its type and value as described in the previous item.

4.10.1.2 Integration of existing Concepts and Support for Services and Plugs

In this section we show items 1. to 3. listed at the end of Section 4.10.

Ad 1 *An implementation of `I_ServiceAccess` can be provided by a special wrapper for atomic UCM-components, if the component of the industrial component model does not implement this interface by itself. That is, the component was not yet designed with respect*

to our component model, but can nevertheless be integrated by only providing a suitable component interface specification and a component implementation specification.

If a JavaBean is unaware of our component model, it will not implement `I_ServiceAccess`. We can nevertheless integrate such a JavaBean by providing a component interface specification which declares all interfaces implemented by the JavaBean as provided services. A wrapper class can be declared which implements `I_ServiceAccess` and which has an attribute of the type of the JavaBean-class. For all services declared as provided ones, this wrapper class returns a reference to its internally stored bean when `getServiceReference` is called. A call to `getConnectionPointObject` results in a return value of `null`. The component implementation specification refers to this wrapper class instead of the original JavaBean.

Ad 2 *Existing approaches for services or event connections can be integrated into our model.*

The most relevant concepts used for the JavaBeans component model are properties and events. Concepts like services and plugs are not supported at all. Nevertheless, event notifications can be integrated into our model as optional required services.

Event sources can be determined by their implemented (de)registration methods for `EventListeners` as described in Section 2.2.1.1 item 3. In the following, these methods are referred to as *add- and removeXYZListener-methods*.

Every JavaBean providing an *add- and removeXYZListener-method* can be regarded as a component having an optional required service of type `XYZListener` with connect-method *addXYZListener* and disconnect-method *removeXYZListener*. Only service providers implementing the corresponding `XYZListener` type can be connected to the event sources. An implemented `EventListener` interface (`XYZListener`) can be regarded as a provided service of type `XYZListener`. So event connections can be regarded as service connections. The *add- and removeXYZListener-methods* are combined into an interface `IConnectionPoint_XYZListener`.

The following example shows how the possibility to fire `MouseEvent`s can be expressed by the declaration of an optional required service of type `java.awt.event.MouseListener`.

```
// Java interface declaring the (de)registration
// methods for MouseListeners
package event_connection_points;
import java.awt.event.*;

public interface IConnectionPoint_MouseListener {
    public void addMouseListener(MouseListener l);
    public void removeMouseListener(MouseListener l);
}
```

```
// Component interface specification
ComponentInterface CI_... {
    GeneralDescriptions
        NamingConventions = JavaType

    ServiceDefinitions
        ProvidedServices
        ...
    RequiredServices
        MouseEventFiring :
            (java.awt.event.MouseListener,
             event_connection_points.IConnectionPoint_MouseListener,
             [0...*])
        ...
    ServiceRelations
        ...
}
```

Ad 3 *Concepts like our services and plugs can be used even if related concepts are missing in the industrial component model under consideration. In addition, service and plug connections can be applied.*

Provided Services: To provide a service, the JavaBean-class has to implement the interface specified as the type of the service or it must provide a reference to an internal object implementing this interface. This reference has to be obtained when calling `getServiceReference` for this service on a JavaBean-class object. Although example 4.1.14 on page 93 only uses pure java classes instead of JavaBeans, this example demonstrates all necessary steps. The classes representing the components in this example correspond to the JavaBean-classes.

Required Services: A required service can be realized by declaring an attribute of the corresponding service interface type in the class realizing the connection point belonging to the required service. This attribute is used to store a reference to an object implementing the methods declared by the service interface type of the required service. The methods used to store or delete such references have to obey the rules for connection point methods from Section 4.1.1.1 on page 92. (See example 4.1.14, too.)

Plugs: Plugs can also be used because they can simply be defined by a suitable component interface specification enumerating the names of the provided and/or required services belonging to the plug. No counterpart is needed in the underlying industrial component model.

Interconnections: Interconnections between required and provided services can be established by calling one of the connect-methods on the connection point object associated with a required service and passing as an argument an object implementing the provided service to be connected to the required one. References to connection point objects and objects implementing the provided service can

be obtained by calling the methods of the service access interface. For examples please refer to example 4.1.14.

4.10.2 Component Object Model (COM)

4.10.2.1 Component Implementations and Component Interface Specifications

As in the case of JavaBeans, here we answer the first four questions from Section 4.10.

Ad 1 *What is a suitable declaration of `I_ServiceAccess`?*

For COM components, the service access interface is declared as follows.

```
class I_ServiceAccess {
public:
    virtual IUnknown* getServiceReference (BSTR PServiceName) = 0;
    virtual IUnknown* getConnectionPointObject (BSTR RServiceName) = 0;
}
```

BSTR is a special string format used in the context of COM.

`I_ServiceAccess` is suitable because `IUnknown` covers all COM interface types.

Ad 2 *How does our component interface specification look like for this industrial component model? Especially: how are service interface types denoted?*

Service interface types for COM components are denoted by their interface identifiers (IIDs). The fact that IIDs are used to denote service interface types must be announced by setting the value for 'NamingConventions' in the component interface specification to 'GUID'.

The following example shows how component interface specifications look like for COM. This example is the COM-version of example 4.1.25. In this example, some globally unique identifiers (GUID) are used to identify COM interfaces. These GUIDs are arbitrarily chosen as shown below. The used interface identifiers are chosen to be:

- {00000001-0000-0000-0000-000000000040} for `I_ServiceAccess`
- {00000001-0000-0000-0000-000000000050} for `I_List`
- {00000001-0000-0000-0000-000000000060} for `I_Order`
- {00000001-0000-0000-0000-000000000070} for `I_Customer`
- {20000001-0000-0000-0000-000000000001} for `I_ConnectionPoint_List`.

```

/***** Component Interface *****/

ComponentInterface CI_OrderAdministration {
    GeneralDescriptions
        NamingConventions = GUID

    ServiceDefinitions
        ProvidedServices
            ServiceAccess      : {00000001-0000-0000-0000-000000000040}
            Orders              : {00000001-0000-0000-0000-000000000060}
            Customers           : {00000001-0000-0000-0000-000000000070}
        RequiredServices
            OrderList           : ({00000001-0000-0000-0000-000000000050},
                                {20000001-0000-0000-0000-000000000001},
                                [1...1])
            CustomerList        : ({00000001-0000-0000-0000-000000000050},
                                {20000001-0000-0000-0000-000000000001},
                                [1...1])

    ServiceRelations
        Constraints
            DifferentLists      = {OrderList, CustomerList}
}

/*****/

```

Subtype relations for service interface types can be determined based on their IIDs. If one wants to know whether $\text{IID}_1 \preceq \text{IID}_2$ holds, one may search the windows registry starting at²⁵ `HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{IID_1}`. If IID_1 is directly derived from IID_2 , then IID_2 is registered as its base interface using the registry subkey `BaseInterface` as shown in Figure 4.35.

The subtype relation holds. If another IID (IID_3) is stored as a base interface for IID_1 one has to proceed by looking for base interfaces of IID_3 and so forth. The search stops, if IID_2 is found in the chain of base interfaces or if no further base interface can be found. In this case $\text{IID}_1 \preceq \text{IID}_2$ does not hold.

Another possibility to determine whether $\text{IID}_1 \preceq \text{IID}_2$ holds is to search the type library containing a description of the interface with IID_1 . For an explanation on how base interfaces may be determined using type libraries please refer to Section 2.2.2.4.

²⁵Several API-functions allow one to access the Windows registry as e.g. `RegOpenKeyEx` and `RegQueryValueEx`.

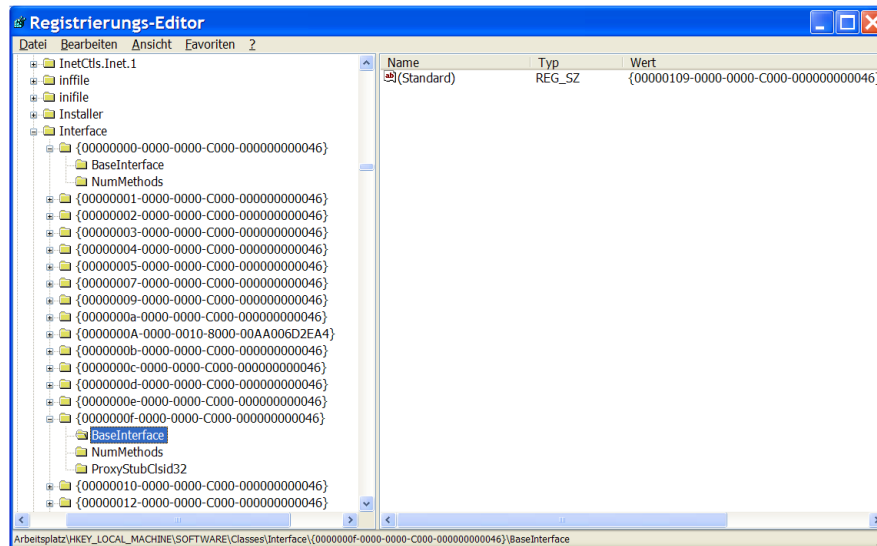


Figure 4.35: Windows Registry for COM Interfaces

Ad 3 *How does a component implementation for an atomic UCM-component look like? Especially: how is a component of the industrial component model referred to and how can an instance be built from this information?*

In component implementations of atomic UCM-components the value for 'ComponentModel' is set to 'COM' and a COM component is referred to by its class ID (CLSID).

For instantiation, `CoCreateInstance` from the COM library can be called using this CLSID while simultaneously querying for a reference to `IUnknown`.

A special wrapper object for atomic UCM-components is generated implementing `I_ServiceAccess` and storing the obtained reference to `IUnknown`. This wrapper could e.g. be declared as follows:

```
class IServiceAccess_COM : public IUnknown {
public:
    virtual HRESULT getServiceReference (BSTR PServiceName,
                                         IUnknown** ppUnknown) = 0;
    virtual HRESULT getConnectionPointObject (BSTR RServiceName,
                                              IUnknown** ppUnknown) = 0;
}

class I_TypeInfo {
public:
    virtual ITypeInfo* getServiceTypeInfo (BSTR PServiceName) = 0;
    virtual ITypeInfo* getConnectionPointTypeInfo (BSTR RServiceName) = 0;
}
```

```

class AtomicComp : public I_ServiceAccess, public I_TypeInfo {
    ...
private:
    ITypeLib *          pTypeLib;
    ITypeInfo *         pTypeInfo;
    IUnknown *          pUnknown;
    IServiceAccess_COM * pServiceAccess;
    ...
}

```

Then `QueryInterface` can be called on `pUnknown` to obtain a reference to `IServiceAccess_COM`. On success this reference is stored in `pServiceAccess`. If the methods of `I_ServiceAccess` are called on the wrapper, it calls the corresponding methods of `pServiceAccess` if non-NULL. Otherwise it calls `QueryInterface` on `pUnknown` to retrieve references to the requested services or connection points. This is only needed for COM components being unaware of our component model. For more information on how to integrate unaware components see Section 4.10.2.2.

To be able to retrieve runtime type information on the COM class and its interfaces, the wrapper also stores a reference to an `ITypeLib` object representing the type library containing information for the specified COM class. This reference can e.g. be obtained by a call to `LoadRegTypeLib` passing the corresponding LIBID as parameter. For information on how this LIBID can be found and for information on the purpose of type libraries please refer to Section 2.2.2.4. The wrapper also stores a reference to an `ITypeInfo` object providing the run time type information for the COM class itself.

```

// Retrieve the LIBID from the CLSID.
// Get a reference to the type library object.
...
unsigned short verMajor = ...;
unsigned short verMinor = ...;
HRESULT hr;
hr = LoadRegTypeLib(LIBID, verMajor, verMinor, LANG_NEUTRAL, &pTypeLib);

.....

// Get a reference to an ITypeInfo object providing the runtime
// type information for the COM class identified by CLSID.
hr = pTypeLib-> GetTypeInfoOfGuid (CLSID, &pTypeInfo);

.....

```

The runtime type information will be needed to build composite UCM-components. Details are described in item Ad 4.

An example of two component implementation specifications for atomic UCM-components is shown below. The example refers to the IIDs and the component interface specification `CI_OrderAdministration` introduced in item Ad 2.

The class IDs for the two COM components providing a list implementation and an order administration are chosen as follows:

- {90000001-0000-0000-0000-000000000001} for the COM component providing a list implementation,
- {90000001-0000-0000-0000-000000000010} for the COM component providing the order administration.

```

/***** Component Interfaces *****/

ComponentInterface CI_List {
    GeneralDescriptions
        NamingConventions = GUID

    ServiceDefinitions
        ProvidedServices
            ServiceAccess      : {00000001-0000-0000-0000-000000000040}
            List                : {00000001-0000-0000-0000-000000000050}
}

/**** Component Implementations for Atomic UCM-Components ****/

Component CP_List implements CI_List {
    GeneralDescriptions
        type                = atomic
        ComponentModel       = COM
        ImplementingComponent = {90000001-0000-0000-0000-000000000001}
        // CLSID for COM component providing a list implementation.
}

Component CP_OrderAdministration implements CI_OrderAdministration
{
    GeneralDescriptions
        type                = atomic
        ComponentModel       = COM
        ImplementingComponent = {90000001-0000-0000-0000-000000000010}
        // CLSID for COM component providing an order administration.
}

/*****/

```

Ad 4 *Can a composite UCM-component be built from its component implementation specification? Especially: Can a connection be established based on the information on the*

connect-method in the component implementation or on the default connect-method in the component interface specification?

An example of a component implementation specification for a composite UCM-component is shown below. The example refers to the IIDs, component interface specifications and component implementations for atomic UCM-components from items Ad 2 and Ad 3.

```

/***** Component Interface Specification *****/

ComponentInterface CI_SpecialOrderAdministration {
  GeneralDescriptions
    NamingConventions = GUID

  ServiceDefinitions
    ProvidedServices
      ServiceAccess      : {00000001-0000-0000-0000-000000000040}
      Orders              : {00000001-0000-0000-0000-000000000060}
      Customers           : {00000001-0000-0000-0000-000000000070}
}

/***** Component Implementations for Composite UCM-Components *****/

Component CP_SpecialOrderAdministration implements
                                     CI_SpecialOrderAdministration {
  GeneralDescriptions
    type                  = composite

  Parts
    P_OrderAdministration : CI_OrderAdministration
    P_OrderList            : CI_List
    P_CustomerList         : CI_List

  InternalConnections
    RequiredServices
      P_OrderAdministration.OrderList
        <== (connect({00000001-0000-0000-0000-000000000050} -,
                    VT_BSTR OrderList))
        P_OrderList.List
      P_OrderAdministration.CustomerList
        <== (connect({00000001-0000-0000-0000-000000000050} -,
                    VT_BSTR CustomerList))
        P_CustomerList.List

  Exports
    ProvidedServices
      CI_SpecialOrderAdministration.Orders
        <-- P_OrderAdministration.Orders
      CI_SpecialOrderAdministration.Customers
        <-- P_OrderAdministration.Customers

```

```

ImplementationBinding
  CI_List <<< CP_List
  CI_OrderAdministration <<< CP_OrderAdministration
}

/*****

```

The primitive data types used as parameter types in connect-method declarations are denoted by enumeration constants for the data types representable by type VARIANT (see paragraph *Type Descriptions for COM Interfaces* on page 47). The data type for the parameter accepting the needed service interface is denoted by its IID.

As in the case of JavaBeans, to build composite UCM-components it only remains to show that it is possible

- to determine a method object representing the connect-method which can then be used to call this method. It must be possible to determine this method object only based on its name and parameter types given as strings.
- to generate arguments for the method call based on the values given as strings. (For the argument with value "-", *servObj* has to be used as actual parameter.)
- to execute the connect-method on the connection point object.

Determine a method object: To be able to achieve this goal for COM, we need the runtime type information for the type of the connection point. This type information contains descriptions for all methods declared for this type. To be able to retrieve this information we require²⁶ that the type library assigned to a COM class also provides type information for all connection point types needed for its required services with one exception: *IConnectionPoint*. This is due to the fact that we want to be able to integrate COM components into our model, even if they are not designed for this purpose. As these components normally provide type information on their provided and outgoing interfaces, but not on *IConnectionPoint*, we can not insist on type information for *IConnectionPoint* being available by the type library for the COM class under consideration. For every outgoing interface, *IConnectionPoint* is used to register suitable listeners for event notification.

To retrieve the type information for a connection point associated with a required service, we use the method `getConnectionPointTypeInfo` declared in *I_Type*

²⁶Here we do not consider the possibility that a COM object can implement the interface *IProvideClassInfo*. This interface allows one to retrieve type information about the COM object from the COM object itself. We assume type information to be provided by type libraries only. For more information on *IProvideClassInfo* please refer to [EE98].

Info from Ad 3. In the following, `I_TypeInfo` is referred to as the *type-info interface*. The implementation of this interface behaves analogously to the service access interface concerning delegation aspects. The type-info interface of a composite UCM-component delegates a call to `getConnectionPointTypeInfo` to the type-info interface of the wrapper stored for the part exporting the required service. The delegation process terminates on a wrapper for an atomic UCM-component.

Having the name of a required service (*rServiceName*), the IID specifying the connection point type of the connection point associated with this required service (*IID_CP*) can be retrieved from the component interface specification associated with the atomic UCM-component. The declaration of the required service contains, amongst others, information on the service interface type and the connection point type, both identified by their IIDs.

The implementation of `getConnectionPointTypeInfo` from class `AtomicComp` which is used to create wrapper instances for atomic UCM-components can use *IID_CP* to retrieve the type information for this connection point:

```
// Get a reference to an ITypeInfo object providing the runtime
// type information for the interface used by the connection point
// object to register service providers.
ITypeInfo * pTypeInfo;
HRESULT hr = pTypeLib->GetTypeInfoOfGuid(IID_CP, &pTypeInfo);
if (hr == S_OK) return pTypeInfo;
return NULL;
...
```

Using this type information, an identifier for the connect-method (*memID*) can be retrieved based on the name of the method.

```
OLECHAR * methodName = ...;
MEMBERID memID;
HRESULT hr = pTypeInfo->GetIDsOfNames(&methodName, 1, &memID);
```

This *memID* is used later on to call the associated method. Before being able to do so, the arguments for the method call have to be created.

Generate arguments: The execution of the connect-method will be done by a call to `ITypeInfo::Invoke` already described in Section 2.2.2.4 on pages 46 and 47. Therefore, every argument has to be of type `VARIANT` (see Section 2.2.2.4). As already mentioned in the beginning of this section the parameter types for the connect-method are denoted by enumeration constants for the data types representable by the `VARIANT` type as e.g. `VT_BOOL`, `VT_BSTR`, `VT_UNKNOWN`...

Every variable *v* of type `VARIANT` has to be initialized by a call to `VariantInit`. Then its field `vt` used to identify the represented data type has to be set to `VT_BSTR`, because our values for the arguments are only available as strings. The field `bstrVal` used for `BSTRs` has to be set to the value for the argument. Then `VariantChangeType` can be called to convert the string representation in *v* to the type specified by the enumeration constant for the argument as e.g. `VT_INT`. An argument value of "-" indicates that the actual parameter is a reference to the object implementing the provided service. In this case `vt` has directly to be set to `VT_UNKNOWN` and the corresponding `VARIANT`-field `pUnkVal` to a reference to the object implementing the provided service, *servObj*:

```
v.vt = VT_UNKNOWN;
v.pUnkVal = (IUnknown*)servObj;
```

Execute the connect-method: The connect-method is executed by `ITypeInfo::Invoke`.

The following code snippet shows the fundamental steps.

```
VARIANT result;
VariantInit(&result);

// Determine the number of arguments to be generated.
// This number can be determined from the string
// representation of the connect-method.
UINT numberParams = ...;

// Declare an array for the arguments.
VARIANTARG args[numberParams];

// Generate all arguments as described above and store them in args.
// Declare a DISPPARAMS variable and assign the generated arguments.
DISPPARAMS params = {args, NULL, numberParams, 0};

// Call Invoke. The first parameter has to be a reference to the
// connection point object. Identify the method to be called by 'memID'
// already determined from the name of the connect-method.
// Pass the generated arguments in params.
hr = pTypeInfo->Invoke( conObj, memID, DISPATCH_METHOD,
                      &params, &result, NULL, NULL);
```

The connection point object *conObj* can be retrieved as shown in Section 4.9.

4.10.2.2 Integration of existing Concepts and Support for Services and Plugs

In this section we show items 1. to 3. listed at the end of Section 4.10.

Ad 1 *An implementation of `I_ServiceAccess` can be provided by a special wrapper for atomic UCM-components, if the component of the industrial component model does not implement this interface by itself. That is, the component was not yet designed with respect to our component model, but can nevertheless be integrated by only providing a suitable component interface specification and a component implementation specification.*

If a COM component is unaware of our component model, it will not implement `I_ServiceAccess_COM`. We can nevertheless integrate such a COM component by providing a component interface specification which declares all interfaces implemented by the COM component as provided services and all outgoing interfaces as optional required services with COM's `IConnectionPoint` interface as connection point type (see below). As described in Section 4.10.2.1 item Ad 3, an instance of a COM component is wrapped by an instance of the class `AtomicComp`, in the following referred to as *wrapper_AtomicComp*. As the COM component does not implement `I_ServiceAccess_COM`, `pServiceAccess` of *wrapper_AtomicComp* is `NULL`.

Querying for provided services: Therefore, if a reference to a provided service is requested on *wrapper_AtomicComp*, it does not delegate this call to `pServiceAccess`. Instead, *wrapper_AtomicComp* retrieves the IID corresponding to the requested service from the component interface specification and calls `QueryInterface` on `pUnknown` passing this IID to retrieve a reference to the requested service.

Querying for connection point objects: As will be shown in more detail in item Ad 2, outgoing interfaces can be treated as optional required services. The type of the required service corresponds to the type of the outgoing interface. The connection point object corresponds to the connection point object declared by COM.

If a reference to a connection point object for a required service is requested by a call to `getConnectionPointObject` on *wrapper_AtomicComp*, the IID identifying the type of the required service respectively outgoing interface can be retrieved from the component interface specification. *wrapper_AtomicComp* first calls `QueryInterface` on `pUnknown` to retrieve a reference to the COM interface `IConnectionPointContainer`. Subsequently, the wrapper calls `FindConnectionPoint` on this interface passing as argument the IID of the required service. It returns the pointer to the object implementing `IConnectionPoint` retrieved from `FindConnectionPoint`. For details on outgoing interfaces see Section 2.2.2.2.

Connections: As arbitrary COM components are not forced to provide runtime type information for the interface `IConnectionPoint`, the procedure from Section 4.10.2.1 item Ad 4 using `ITypeInfo` objects to establish connections can not be used.

To determine whether `IConnectionPoint` is used as the connection point type for a required service, the IID of its connection point (*IID_CP*) is determined from the component interface specification. If *IID_CP* equals *IID_IConnectionPoint* (B196B286-BAB4-101A-B69C-00AA00341D07), the connection can be established by the following steps:

```

IConnectionPoint * pConnectionPoint;
DWORD dwCookie;

pConnectionPoint = (IConnectionPoint*)(
    wrapper_AtomicComp.getConnectionPoint(rServiceName)
);

// Establishing the connection using the standard method 'Advise'
pConnectionPoint->Advise((IUnknown*)(
    wrapper_AtomicComp.getServiceReference
        (pServiceName)
),
    &dwCookie
);

```

Ad 2 *Existing approaches for services or event connections can be integrated into our model.*

Interfaces provided by a COM component can be used as provided services in which the type of the service is identified by the IID of the COM interface.

Event connections using outgoing interfaces can be treated as optional required services. The type of the required service corresponds to the type of the outgoing interface. The connection point object corresponds to the connection point object declared by COM. All connection point objects in COM for outgoing interfaces have to implement the interface `IConnectionPoint` which declares standard methods to register and deregister interested listeners (`Advise`, `Unadvise`). That is, for every outgoing interface the type of its connection point is `IConnectionPoint`. For details on outgoing interfaces see Section 2.2.2.2. Therefore, outgoing interfaces are typically declared in our component interface specifications as follows:

```

ComponentInterface CI_... {
    GeneralDescriptions
        NamingConventions = GUID

    ServiceDefinitions
        ProvidedServices
            ...

    RequiredServices
        ExampleName :
            ( {CA000001-0000-0000-0000-000000000009}, // IID of outgoing interface
              {B196B286-BAB4-101A-B69C-00AA00341D07}, // IID of IConnectionPoint
              [0...*]
            )

```

```

        )
    ...

    ServiceRelations
    ...
}

```

In the pattern from above *ExampleName* is exemplarily used as the name of a required service representing an arbitrary outgoing interface. {CA000001-0000-0000-0000-000000000099} is used as the IID of the outgoing interface representing the type of the required service. The service name and the IID will vary depending on the actual component interface specification and the type of the outgoing interface. Some outgoing interfaces do not support the registering of an unlimited number of listeners. In these cases '*' from [0 . . . *] has to be set to the upper limit on simultaneously registered listeners.

Ad 3 *Concepts like our services and plugs can be used even if related concepts are missing in the industrial component model under consideration. In addition, service and plug connections can be applied.*

COM does not directly support the concept of a required service or a plug.

Although outgoing interfaces can be used as optional required services, other required services can be declared using their own connection point interfaces. For these required services, it is not necessary that the COM component implements `IConnectionPointContainer`. Only `IServiceAccess_COM` has to be implemented to be able to retrieve the connection point object for a required service. Thus it is easier and more efficient to establish a connection using this way. Plugs can also be used because they can simply be defined by a suitable component interface specification. No counterpart is needed in the underlying industrial component model.

4.10.3 .NET

4.10.3.1 Component Implementations and Component Interfaces Specifications

As for JavaBeans and COM, in this subsection we answer the first four questions from Section 4.10.

We start with an example of component interface and implementation specifications using .NET components. This example is the .NET pendant to the examples given for JavaBeans and COM components. We shall refer to this example to discuss the entries relevant to Ad 1 - Ad 4. In all examples used in this section, assemblies are assumed to reside in the same application directory. Thus the assembly text name is sufficient to identify an assembly. For more details on assembly names please refer to the MSDN-Library. We use C# as programming language for code snippets.

Example 4.10.1 (Component Implementations and Interface Specifications for .NET)

The following assumptions are made in the context of this example:

- *The class providing the list implementation resides in the assembly ListImplementations and the class implementing the order administration resides in the assembly OrderAdministration.*
- *Component implementations are grouped in the Components namespace.*
- *Service interfaces are grouped in the namespace Components.ServiceInterfaces and are defined in the assembly AssemblyServiceInterfaces.*
- *Connection point interfaces are grouped in the namespace Components.ConnectionPoints and are defined in the assembly AssemblyConnectionPoints.*

```

/***** Component Interfaces *****/

ComponentInterface CI_List {
    GeneralDescriptions
        NamingConventions = .NETType

    ServiceDefinitions
        ProvidedServices
            ServiceAccess      :
                'Components.ServiceInterfaces.I_ServiceAccess,AssemblyServiceInterfaces'
            List                :
                'Components.ServiceInterfaces.I_List,AssemblyServiceInterfaces'
}

ComponentInterface CI_OrderAdministration {
    GeneralDescriptions
        NamingConventions = .NETType

    ServiceDefinitions
        ProvidedServices
            ServiceAccess      :
                'Components.ServiceInterfaces.I_ServiceAccess,AssemblyServiceInterfaces'
            Orders              :
                'Components.ServiceInterfaces.I_Order,AssemblyServiceInterfaces'
            Customers           :
                'Components.ServiceInterfaces.I_Customer,AssemblyServiceInterfaces'

    RequiredServices
        OrderList              :
            ( 'Components.ServiceInterfaces.I_List,AssemblyServiceInterfaces',
              'Components.ConnectionPoints.IConnectionPoint_List,AssemblyConnectionPoints'
              [1...1])
        CustomerList           :

```

```

        ('Components.ServiceInterfaces.I_List,AssemblyServiceInterfaces',
         'Components.ConnectionPoints.IConnectionPoint_List,AssemblyConnectionPoints'
         [1...1])

ServiceRelations
  Constraints
    DifferentLists = {OrderList, CustomerList}
}

ComponentInterface CI_SpecialOrderAdministration {
  GeneralDescriptions
    NamingConventions = .NETType

  ServiceDefinitions
    ProvidedServices
      ServiceAccess :
        'Components.ServiceInterfaces.I_ServiceAccess,AssemblyServiceInterfaces'
      Orders :
        'Components.ServiceInterfaces.I_Order,AssemblyServiceInterfaces'
      Customers :
        'Components.ServiceInterfaces.I_Customer,AssemblyServiceInterfaces'
}

/***** Component Implementations for Atomic UCM-Components *****/

Component CP_List implements CI_List {
  GeneralDescriptions
    type = atomic
    ComponentModel = .NET
    ImplementingComponent = 'Components.List,ListImplementations'
    // .NET class providing a list implementation and residing
    // in assembly ListImplementations
}

Component CP_OrderAdministration implements CI_OrderAdministration
{
  GeneralDescriptions
    type = atomic
    ComponentModel = .NET
    ImplementingComponent = 'Components.OrderAdministration,OrderAdministration'
    // .NET class providing the implementation and residing
    // in assembly OrderAdministration
}

/***** Component Implementations for Composite UCM-Components *****/

Component CP_SpecialOrderAdministration implements
    CI_SpecialOrderAdministration {
  GeneralDescriptions

```

```

type                                = composite

Parts
  P_OrderAdministration : CI_OrderAdministration
  P_OrderList           : CI_List
  P_CustomerList        : CI_List

InternalConnections
  RequiredServices
    P_OrderAdministration.OrderList
      <== (connect(
        'Components.ServiceInterfaces.I_List,AssemblyServiceInterfaces' -,
        'System.String' OrderList))
    P_OrderList.List
  P_OrderAdministration.CustomerList
      <== (connect(
        'Components.ServiceInterfaces.I_List,AssemblyServiceInterfaces' -,
        'System.String' CustomerList))
    P_CustomerList.List

Exports
  ProvidedServices
    CI_SpecialOrderAdministration.Orders
      <-- P_OrderAdministration.Orders
    CI_SpecialOrderAdministration.Customers
      <-- P_OrderAdministration.Customers

ImplementationBinding
  CI_List <<< CP_List
  CI_OrderAdministration <<< CP_OrderAdministration
}

```

Ad 1 *What is a suitable declaration of I_ServiceAccess?*

In C# I_ServiceAccess is declared as:

```

interface I_ServiceAccess {
  object getServiceReference (string PServiceName);
  object getConnectionPointObject (string RServiceName);
}

```

string is an alias for System.String and object for System.Object.

References for all types of interfaces and connection points can be obtained by calls to the methods of I_ServiceAccess since object covers all these types.

Ad 2 *How does our component interface specification look like for this industrial component model? Especially: how are service interface types denoted?*

Service interface types for .NET components are denoted by the names of their corresponding .NET interfaces. .NET interface names are themselves denoted by their fully qualified names that is, including namespace information and, if necessary, the name of the assembly containing the type. The fact that .NET types are used to denote service interface types must be announced by setting the value for 'NamingConventions' in the component interface specification to '.NETType'. For examples please refer to example 4.10.1.

Subtype relations for service interface types can be determined as shown below, based on the names of their .NET types being available as strings from the component interface specification. The `Type` object corresponding to the service interface expected to be a subtype of the other one can be obtained by calling `Type.GetType` passing the string representation of the .NET type of this interface as argument. On this `Type` object one can call `GetInterface` with signature `public Type GetInterface(string name);`. As parameter one passes the string representation of the interface expected to be a supertype of the former. If this method returns a non null reference, the expected subtype relationship between both interfaces holds. In the following, the method `IsSuperinterface` checks whether its first argument is a subtype of its second argument.

```
// Returns 'true', if the second argument is a supertype of
// the first one. Otherwise, 'false' is returned.
private bool IsSuperinterface (string subInterface,
                               string superInterface) {
    Type typeSub = Type.GetType(subInterface);
    if (typeSub != null) {
        Type type = typeSub.GetInterface(superInterface);
        if (type != null)
            return true;
        else return false;
    }
    else return false;
}
```

Ad 3 *How does a component implementation for an atomic UCM-component look like? Especially: how is a component of the industrial component model referred to and how can an instance be built from this information?*

In component implementations of atomic UCM-components the value for 'ComponentModel' is set to '.NET' and a .NET component is referred to by its class name. .NET class names are denoted by their fully qualified names that is, including namespace information and, if necessary, the name of the assembly containing the type.

An instance of the .NET component can be created by first loading the assembly containing the .NET class by calling `Assembly.Load` and then instantiating the

class by calling `assembly.CreateInstance` on the assembly object obtained by the Load operation (see Section 2.2.3.1 item 4 on page 58). Another possibility is to first retrieve a `Type` object `typeObj` for the class by calling `Type.GetType` and then creating an instance of the class by calling `System.Activator.CreateInstance(typeObj)`; (see Section 2.2.3.4).

For examples of component implementations for atomic UCM-components using .NET components refer to example 4.10.1.

Ad 4 *Can a composite UCM-component be built from its component implementation specification? Especially: Can a connection be established based on the information on the connect-method in the component implementation or on the default connect-method in the component interface specification?*

For examples of component implementation specifications for composite UCM-components using .NET components refer to example 4.10.1.

The data types used as parameter types in connect-method declarations are denoted by their string representation like `'System.String'`, `'System.Int32'`, `'Components.ServiceInterfaces.IList,AssemblyServiceInterfaces'` etc. The assembly name succeeds the interface or class name separated by a comma. For standard data types like e.g. `System.String` the assembly name can be omitted. The .NET API uses this kind of qualified name representation.

As in the case of JavaBeans and COM components, to build composite UCM-components it only remains to show that it is possible

- to determine a method object representing the connect-method which can then be used to call this method. It must be possible to determine this method object only based on its name and parameter types given as strings.
- to generate arguments for the method call based on the values given as strings. (For the argument with value `"-"`, `servObj` has to be used as actual parameter.)
- to execute the connect-method on the connection point object.

Determine a method object: From the parameter types given as strings the corresponding `Type` objects representing their types can be obtained by a call to `Type.GetType` (see Section 2.2.3.4). These `Type` objects can be stored in an array listing all parameter types (e.g. `Type[] parameterTypes`).

The connection point type the connect-method belongs to can be retrieved from the connection point object `conObj` by calling `conObj.GetType()`. Having the `Type` object representing the connection point type one can call `GetMethod` on this `Type` object to retrieve the method object representing the connect-method. The method object is of type `System.Reflection.MethodInfo` and is identified by the method name available as a string and the parameter types available by an array of `Type` objects:

```
MethodInfo conMeth = conObj.GetType().GetMethod(methodName, parameterTypes);
```

Before being able to execute the connect-method, we must be able to generate arguments for the method call based on the values given as strings.

Generate arguments: Because we restricted the argument types for connect-methods to primitive data types and the type of the corresponding required service, one can generate the arguments as follows. Arguments for primitive data types like `System.Boolean`, `System.Byte`, `System.SByte`, `System.Int16`, `System.UInt16`, `System.Int32`, `System.UInt32`, `System.Int64`, `System.UInt64`, `System.Char`, `System.Double`, and `System.Single` can be generated using the methods of class `System.Convert` having a single parameter of type string like

```
public static bool27 ToBoolean(string);           or
public static double ToDouble(string);
```

The following example demonstrates the use of these methods.

```
public object CreateObjectFromString(string strType, string value)
{
    switch (strType) {
        case "System.Boolean": return System.Convert.ToBoolean(value);
        case "System.Int16"   : return System.Convert.ToInt16(value);
        case "System.Int64"   : return System.Convert.ToInt64(value);
        // Code for other primitive data types ...
        default : return null;
    }
}
```

An argument value of "-" for an argument of the connect-method indicates that the actual parameter is a reference to the object implementing the provided service. Therefore this argument can be set to *servObj*.

Execute the connect-method: As already mentioned in Section 2.2.3.4, a method represented by an object of type `MethodInfo` can be executed by calling `Invoke` on this object. The first parameter of `Invoke` refers to the object the method is invoked from. In our case this is *conObj*. The second parameter refers to an array of objects representing the arguments of the method to be executed by `Invoke`. This procedure is analogous to the one for JavaBeans.

4.10.3.2 Integration of existing Concepts and Support for Services and Plugs

In this section we show items 1. to 3. listed at the end of Section 4.10.

Ad 1 *An implementation of `I.ServiceAccess` can be provided by a special wrapper for atomic UCM-components, if the component of the industrial component model does not implement this interface by itself. That is, the component was not yet designed with respect*

²⁷`bool` is an alias for `System.Boolean`.

to our component model, but can nevertheless be integrated by only providing a suitable component interface specification and a component implementation specification.

If a .NET component is unaware of our component model, it will not implement `I_ServiceAccess`. We can nevertheless integrate such a .NET component by the same means as already described for JavaBeans.

Ad 2 *Existing approaches for services or event connections can be integrated into our model.*

The most relevant concepts provided for .NET components as defined in this thesis are properties and events. Concepts like services and plugs are not supported at all. .NET events cannot directly be integrated as subscribing occurs on method-level instead of interface-level. Nevertheless, event registration can be raised to the level of interfaces using the mechanisms provided by the JavaBeans component model. Several related event notification methods are grouped by an interface, in the following referred to as (event-) interface. Objects interested in a notification have to implement this interface. A publisher of the events belonging to the (event-) interface has to provide a registration method which takes as its argument a subscriber object having as type the type of the (event-) interface. The following example demonstrates this technique.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace EventInterfaces {

    /******* Interface grouping mouse events *****/
    public interface IMouseEvents {
        void OnMouseClicked();
        void OnMouseMove();
        void OnMouseOver();
        // ...
    }

    /******* Delegate type for mouse events *****/
    public delegate void MouseEvent();

    /******* Publisher for mouse events *****/
    public class MouseEventSource {

        private event MouseEvent m_MouseClick;
        private event MouseEvent m_MouseMove;
        private event MouseEvent m_MouseOver;
```

```

// ...

public void FireMouseClicked() {
    if (m_MouseClick != null)
        m_MouseClick();
}

public void FireMouseMove() {
    if (m_MouseMove != null)
        m_MouseMove();
}

public void FireMouseOver() {
    if (m_MouseOver != null)
        m_MouseOver();
}

//...

public void Subscribe (IMouseEvents mouseSubscriber) {
    m_MouseClick += mouseSubscriber.OnMouseClicked;
    m_MouseMove  += mouseSubscriber.OnMouseMove;
    m_MouseOver  += mouseSubscriber.OnMouseOver;
    // ...
}

public void Unsubscribe(IMouseEvents mouseSubscriber) {
    m_MouseClick -= mouseSubscriber.OnMouseClicked;
    m_MouseMove  -= mouseSubscriber.OnMouseMove;
    m_MouseOver  -= mouseSubscriber.OnMouseOver;
    // ...
}
}

/***** Consumer of mouse events *****/
public class Form1 : Form, IMouseEvents {
    // Declaration of labels and buttons
    ...

    /**** Event source / publisher ****/
    MouseEventSource m_MouseEventSource = new MouseEventSource();

    private void InitializeComponent() {
        // Add labels and buttons to form-window,
        // register event handler for buttons ...
    }

    public Form1()
    {
        InitializeComponent();
    }
}

```



```

        /**** Subscribe to event source ****/
        m_MouseEventSource.Subscribe(this);
    }

    public void OnMouseClicked() {
        label.Text = "Mouse-CLICK-Event occurred";
    }

    public void OnMouseMove() {
        label.Text = "Mouse-MOVE-Event occurred";
    }

    public void OnMouseOver() {
        label.Text = "Mouse-OVER-Event occurred";
    }

    private void button1_Click(object sender, EventArgs e) {
        m_MouseEventSource.FireMouseClicked();
    }

    private void button2_Click(object sender, EventArgs e) {
        m_MouseEventSource.FireMouseMove();
    }

    private void button3_Click(object sender, EventArgs e) {
        m_MouseEventSource.FireMouseOver();
    }
}
}

```

The possibility of a component to fire events grouped by an (event-) interface can be expressed in our model by an optional required service having as service interface type the type of the (event-) interface. The (de)registration methods have to be grouped in a corresponding connection point interface which has to be implemented by the event source (in our example `MouseEventSource`). This technique is outlined by the following example. This example assumes the .NET interfaces to be declared in assembly `AssemblyEventHandling`.

```

// .NET interface declaring the (de)registration
// methods for consumers of mouse events resp.
// subscribers to mouse events
namespace EventInterfaces {

    public interface IConnectionPoint_MouseEvents {
        public void Subscribe (IMouseEvents mouseSubscriber);
        public void Unsubscribe(IMouseEvents mouseSubscriber);
    }
}

```

```

}

// Component interface specification
ComponentInterface CI_... {
    GeneralDescriptions
        NamingConventions = .NETType

    ServiceDefinitions
        ProvidedServices
        ...
        RequiredServices
            MouseEventFiring :
                ( 'EventInterfaces.IMouseEvents,AssemblyEventHandling',
                  'EventInterfaces.IConnectionPoint_MouseEvents,
                                                            AssemblyEventHandling',
                  [0...*])
            ...
    ServiceRelations
        ...
}

```

Ad 3 *Concepts like our services and plugs can be used even if related concepts are missing in the industrial component model under consideration. In addition, service and plug connections can be applied.*

This can be achieved by the same means as for JavaBeans and will not be repeated here. The only thing stressed for .NET is the possibility to explicitly implement interfaces which are types of provided services. This enables stronger encapsulation as in the case of JavaBeans. For more details on explicitly implemented interfaces please refer to Section 2.2.3.1 on page 50.

4.10.4 Comparing Integration for the Various Component Models

Component interface and implementation specifications only vary slightly depending on the industrial component model integrated. The following table lists where differences appear in these specifications and where the implementation of the framework differs, e.g. when instantiating atomic UCM-components, checking for a subtype relationship between service interfaces or establishing connections.

Task	JavaBeans	COM components	.NET components
Specification of I_ServiceAccess	Specified as Java interface as shown in Section 4.1.1.1 on page 92.	Specified as C++ pure abstract class as shown in Section 4.10.2.1, item Ad 1.	Specified as C# interface as shown in Section 4.10.3.1, item Ad 1.

Task	JavaBeans	COM components	.NET components
Component interface specification: Setting the value for 'NamingConventions'	The value is set to 'JavaType'.	The value is set to 'GUID'.	The value is set to '.NETType'.
Component interface specification: Specification of service interface types	Specified as Java interface names.	Specified through IIDs.	Specified as .NET interface names.
Checking service interface types for a subtype relationship	Converting service interface names from strings to Class objects using Java reflection. Checking the Class objects for a subtype relationship using method <code>isSubtype</code> as described in Section 4.10.1.1, item Ad 2.	Searching the COM registry for base interfaces or checking the type library as described in Section 4.10.2.1, item Ad 2.	Converting service interface names from strings to Type objects using .NET reflection. Checking the Type objects as shown in method <code>isSuperinterface</code> from Section 4.10.3.1, item Ad 2.
Component implementation for atomic UCM-components: Setting the value for 'ComponentModel'	The value is set to 'JavaBeans'.	The value is set to 'COM'.	The value is set to '.NET'.
Component implementation for atomic UCM-components: Setting the value for 'ImplementingComponent'	The value is set to the name of the JavaBean-class.	The value is set to the class ID of the COM component.	The value is set to the name of the .NET component-class.
Creating component instances for the industrial component models	<code>java.beans.Beans.instantiate</code>	<code>CoCreateInstance</code>	<code>Assembly.Load</code> and in turn <code>assembly.CreateInstance</code>

Task	JavaBeans	COM components	.NET components
Component implementation for composite UCM-components: Specification of the parameter types in connect-methods (Except the connect-method specification, the specification of composite UCM-components does not differ for the various component models.)	The parameter types are denoted as Java class and interface names.	All parameters are of type VARIANT except the parameter used to accept the service object which is denoted by the IID of the COM interface it implements.	The parameter types are denoted as .NET class and interface names.
Establishing a connection	A connection is established using Java reflection as shown in Section 4.10.1.1, item Ad 4.	A connection is established using type libraries as shown in Section 4.10.2.1, item Ad 4.	A connection is established using .NET reflection as shown in Section 4.10.3.1, item Ad 4.
Type of the component implementation object (see Section 4.9)	I.ServiceAccess	I.ServiceAccess and I.TypeInfo	I.ServiceAccess

4.11 Some Algorithms Supporting Visual Composition

In this section, we present some algorithms which enable automatic interconnections between plugs and the composition of plugs to greater ones. These algorithms are especially useful for visual composition and complete the tool support described in Sections 1.1 and 5.

If a plug groups more than one required and one provided service, the determination of complementary plugs and an automated service mapping which is required to connect two plugs automatically is rather difficult. Two algorithms which master this problem are described in the next section. Section 4.11.2 then sketches an algorithm which can be used to link a plug of a component interface *CI* to smaller plugs and single services of constituents of the composite component implementing *CI*.

4.11.1 Automatic Interconnections Using Plugs

When we introduced the term plug (term 4.1.15) we mentioned that plugs constitute units for interconnections between two component instances. Having this concept is already inherently an advantage, because a compiler or tool can then check intercon-

nections between component instances via plugs for correctness. Even if the interconnections are done on service level, it can be checked, whether the services belonging to one plug are only connected to corresponding services of a complementary plug and not to stand-alone services of maybe even different component instances. To ensure the integrity of a plug, all services belonging to one plug have to be connected to corresponding services of one single “partner component” instance. These services have also to be grouped by a plug to ensure that both components agree on the same communication protocol.

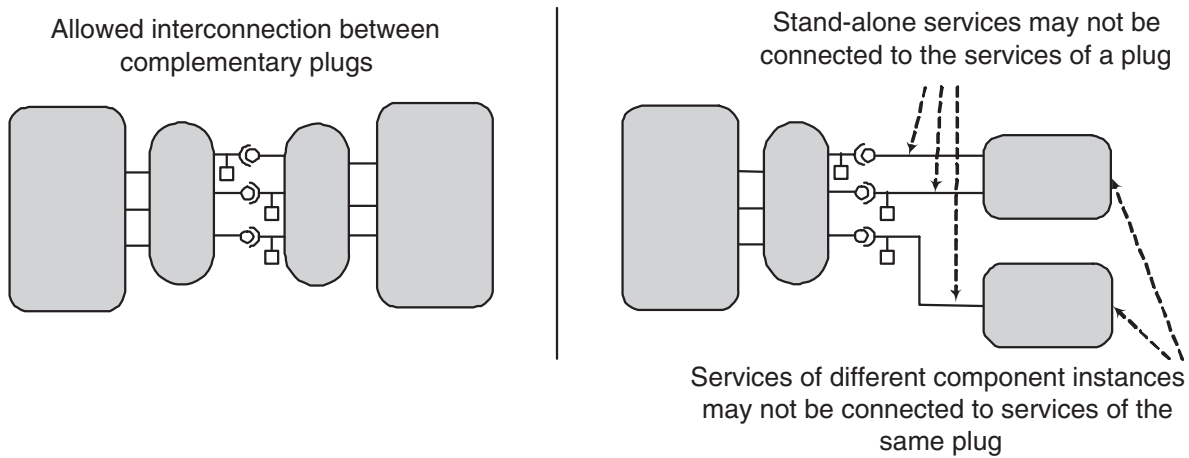


Figure 4.36: Examples for (not) allowed Interconnections

Knowing whether two plugs can be connected or not is already useful for a user of a tool. But he profits even more, if the tool is additionally capable to connect two plugs Pl_1 and Pl_2 automatically that is, the tool need not ask the user for each individual pair of services to be connected.

From theorem 4.4.8 we know that complementary plugs can be connected to each other thereby retaining the integrity condition for plugs. Thus if a tool is able to detect, whether two plugs are complementary, we know that they can be connected. This capability of a tool can also be used to search for all plugs which are complementary to a selected plug. Thus, if a user selects a plug of a component instance residing in the composition window of a tool and wants to know which other plugs from other component instances residing in the composition window can be selected for interconnection, the tool can provide the set of complementary plugs found. Then each plug belonging to this set can be used to establish a valid connection.

Therefore, we first look for an algorithm which tells us, whether two plugs are complementary or not. If they are, we need an algorithm determining a one-to-one mapping from the set of provided services of one plug (Pl_1) to the set of required services of the other plug (Pl_2) and vice versa to actually connect both plugs.

Similar to stand-alone matching services of two components, we allow complementary plugs to have different names and to use different names for their complementary

(matching) provided and required services. One reason for allowing different names is that several constituents of a composite UCM-component can expose the same service or plug. As service and plug-names of the composite component have to be unique, some of them have to be renamed in the component interface of the composite component. Another reason is that two complementary services with service interface types which are not identical will normally reflect this fact by using different service names.

Since we can not rely on equally named services, we have to map complementary services only based on their types.

In the following two subsections, Pl_1 is a plug of a component interface CI_1 and Pl_2 a plug of a component interface CI_2 . For $i = 1$ and $i = 2$, $Required(Pl_i)$ and $Provided(Pl_i)$ are defined as

$$Required(Pl_i) = \{(R, I_R) \mid R \in Required(CI_i, Pl_i) \text{ and } I_R \text{ is the service interface type belonging to } R\}$$

$$Provided(Pl_i) = \{(P, I_P) \mid P \in Provided(CI_i, Pl_i) \text{ and } I_P \text{ is the service interface type belonging to } P\}$$

To simplify the description of the following algorithms we identify service interface types by a name e.g. I_S and do not distinguish between the name and the set of methods $Itype(I_S)$ making up the type.

4.11.1.1 Checking for Complementary Plugs

In this section we introduce an algorithm which is capable to check whether two plugs are complementary. In Section 4.11.1.2 this algorithm will be refined to an algorithm which can be used to (semi-)automatically determine a one-to-one mapping between complementary sets of provided and required services of two plugs.

Sketch of the algorithm: The idea is to use a similar mechanism as to resolve a method call in the case of an overloaded method.

The required services of one plug (e.g. $Required(Pl_2)$) are treated as formal parameters of an overloaded method declaration. The overloaded methods are generated by determining all permutations of the required services. If a permutation appears, in which the types of all parameters are equal to an already existing permutation, the new one is discarded.

The types of the complementary provided services (e.g. $Provided(Pl_1)$) are treated as the types of the actual parameters when calling one of the overloaded methods. For this purpose, the provided services are arranged in an arbitrary order. The algorithm of service-mapping is reduced to the selection of an applicable method of the Java language specification. It is not a problem if there are several applicable methods, since at this stage we only want to decide whether we can find sets of matching services as required. We do not yet look for a one-to-one mapping between services as is necessary, if a connection has to be established.

It follows a more precise description of the algorithm split up into five steps.

Step one: The cardinalities of the complementary sets of services of Pl_1 and Pl_2 are compared to each other. That is, it has to be checked, whether $|Provided(Pl_1)| = |Required(Pl_2)|$ and $|Provided(Pl_2)| = |Required(Pl_1)|$. If the corresponding cardinalities are not equal, Pl_1 and Pl_2 are not complementary plugs and the algorithm stops²⁸. Otherwise it proceeds with step two.

Step two: Let $RS \in \{Required(Pl_1), Required(Pl_2)\}$, $|RS| = n$ and R_i ($1 \leq i \leq n$) the names of the required services belonging to RS with corresponding service interface types I_R_i . Declare a method named m with formal parameters $R_1 \dots R_n$ like

$$void\ m\ (I_R_1\ R_1, I_R_2\ R_2, \dots, I_R_n\ R_n).$$

Generate new method signatures for m by determining all permutations of the required services:

$$void\ m\ (I_R_2\ R_2, I_R_1\ R_1, \dots, I_R_n\ R_n)$$

and so on. If a permutation appears, where the types of all parameters are equal to an already existing method declaration, the new one is discarded.

Step three: Let $PS \in \{Provided(Pl_2), Provided(Pl_1)\}$ be the complementary set of services to RS . Then $|PS| = n$. Let P_i ($1 \leq i \leq n$) be the names of the provided services of PS and I_P_i their corresponding service interface types. Arrange the names of the provided services in an arbitrary order, let us say (P_1, P_2, \dots, P_n) . Treat the sequence of corresponding service interface types $(I_P_1, I_P_2, \dots, I_P_n)$ as the types of the actual parameters when calling one of the overloaded methods m . The algorithm of checking for pairs of matching services can now be reduced to the selection of an applicable method of the Java language specification (step four).

Step four: The sequence of types of the provided services are treated like the types of the argument expressions of an actual call of the method m . The sequence is used to locate method declarations that are applicable, that is, declarations that can be correctly invoked on the given arguments. A method declaration is applicable to a method invocation in our context, if and only if the type of each actual argument is equal to or a subtype of the type of the corresponding parameter. That is, if we have a method declaration

$$void\ m\ (I_R_{j_1}\ R_{j_1}, I_R_{j_2}\ R_{j_2}, \dots, I_R_{j_n}\ R_{j_n})$$

it is applicable, if $\forall i, 1 \leq i \leq n : I_P_i \preceq I_R_{j_i}$.

If there is no method declaration which is applicable, the algorithm stops with an error. More than one method declaration may be applicable to a method invocation without causing an error. For the purpose of deciding whether two plugs are complementary it is enough to know whether there is at least one applicable method declaration²⁹. In this case the algorithm proceeds with step five.

Step five: Steps two to four are repeated for the pair of complementary sets RS and PS not already selected in step two and three during the first run. E.g. if RS and

²⁸See remarks to term 4.4.7.

²⁹A similar algorithm is described in the Java language specification to solve method calls in the context of overloaded methods.

PS were selected to be $Required(Pl_1)$ and $Provided(Pl_2)$ in the first run, in the second run RS will be $Required(Pl_2)$ and PS $Provided(Pl_1)$. For this second run we use a method name m_2 instead of m for the set of overloaded methods generated for this second set of required services RS . If at least one applicable method can be found the algorithm succeeds. In this case we could determine pairs of matching services for both combinations RS and PS from Pl_1 and Pl_2 . This is sufficient to show that both plugs are complementary.

Example 4.11.1 (Checking whether two plugs are complementary) In the following example it will be checked whether the plug from example 4.1.23 on page 113 is complementary to the plugs of Client 1 (part a)) and Client 2 (part b)). The service interface type I_Data used by Client 2 is declared as

interface I_Data extends $I_DataLogging$, $I_ErrorMessages$ {...}
and is therefore a subtype of $I_DataLogging$ and $I_ErrorMessages$.

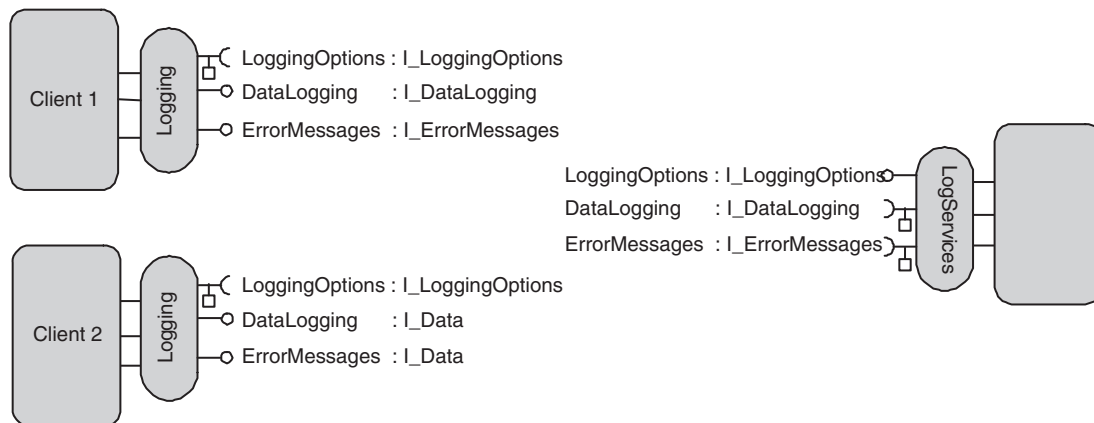


Figure 4.37: Example for Plugs to be checked whether they are complementary

Part a): First, we shall check whether the plug *LogServices* is complementary to the plug *Logging* of Client 1 using the algorithm described above.

Step one: Check cardinalities:

$Provided(LogServices) = \{(LoggingOptions, I_LoggingOptions)\}$ and

$Required(Logging) = \{(LoggingOptions, I_LoggingOptions)\}$.

Therefore, $|Provided(LogServices)| = 1 = |Required(Logging)|$.

Similarly, $Provided(Logging) = \{(DataLogging, I_DataLogging), (ErrorMessages, I_ErrorMessages)\}$
 $= Required(LogServices)$.

Therefore $|Provided(Logging)| = 2 = |Required(LogServices)|$. The cardinality conditions are satisfied and we proceed with step two.

Step two: Generate method signatures:

For $RS = Required(LogServices)$ the following methods are generated by permutation over

the two required services:

void m (I_DataLogging DataLogging, I_ErrorMessages ErrorMessage)

void m (I_ErrorMessages ErrorMessage, I_DataLogging DataLogging)

Step three: Determine a sequence of actual parameter types:

The complementary set of services to RS is $PS = \text{Provided}(\text{Logging})$. We arrange the services belonging to PS as follows: $(\text{DataLogging}, \text{ErrorMessage})$. That is, $\text{DataLogging} = P_1$ and $\text{ErrorMessage} = P_2$. Treat the sequence of corresponding service interface types $(I_DataLogging, I_ErrorMessage)$ as the types of the actual parameters when calling one of the overloaded methods m .

Step four: Looking for applicable methods:

Now we have to determine at least one method declaration that is applicable. Therefore we have to test whether there exists a method *void m (I_R₁ R₁, I_R₂ R₂)* with $R_1 \in \{\text{DataLogging}, \text{ErrorMessage}\}$ and $R_2 \in \{\text{DataLogging}, \text{ErrorMessage}\} \setminus \{R_1\}$ which is applicable, that is $\forall i, 1 \leq i \leq 2 : I_P_i \preceq I_R_i$. Since I_P_1 was selected to be $I_DataLogging$ and I_P_2 to be $I_ErrorMessage$, the only applicable method is

void m (I_DataLogging DataLogging, I_ErrorMessages ErrorMessage).

Step five: Steps two to four are repeated for the pair of complementary sets $RS = \text{Required}(\text{Logging}) = \{(\text{LoggingOptions}, I_LoggingOptions)\}$ and $PS = \text{Provided}(\text{LogServices}) = \{(\text{LoggingOptions}, I_LoggingOptions)\}$. As both sets only consist of one service with identical service interface type, $I_LoggingOptions$, the only generated method

void m_2 (I_LoggingOptions LoggingOptions)

is applicable and the algorithm succeeds. Therefore the plugs *LogServices* and *Logging* of Client 1 are complementary.

Part b): Now we shall check whether the plug *LogServices* is also complementary to the plug *Logging* of Client 2.

Steps one and two of the algorithm are the same as in the previous example.

Step three: Determine a sequence of actual parameter types:

The sequence of services is the same as in part a), but the sequence of corresponding service interface types is (I_Data, I_Data) .

Step four: Looking for applicable methods:

Since $I_Data \preceq I_DataLogging$ and $I_Data \preceq I_ErrorMessage$, both methods generated in step two,

void m (I_DataLogging DataLogging, I_ErrorMessages ErrorMessage),

void m (I_ErrorMessages ErrorMessage, I_DataLogging DataLogging),

are applicable.

Step five of the algorithm is the same as in the previous example. Therefore, the plugs *LogServices* and *Logging* of Client 2 are also complementary.

4.11.1.2 Service Mapping

As we can see from example 4.11.1 part b), it is possible that more than one applicable method may be found. If a user determines in advance that this would be no problem and a mapping could be generated using an arbitrary applicable method, it is sufficient to extend the existing algorithm as follows. Step four is refined by selecting e.g. the first applicable method. If this method has the signature

$$\text{void } m(I_{R_{j_1}} R_{j_1}, I_{R_{j_2}} R_{j_2}, \dots, I_{R_{j_n}} R_{j_n})$$

then a mapping $f: \{P_1, \dots, P_n\} \rightarrow \{R_1, \dots, R_n\}$ is generated with $f(P_i) = R_{j_i} \forall i, 1 \leq i \leq n$.

Example 4.11.2 (Automatically resolving Ambiguity) In example 4.11.1 part b) we selected the following sequence of provided services belonging to *Provided(Logging)*:

(*DataLogging*, *ErrorMessages*), that is, $P_1 = \text{DataLogging}$ and $P_2 = \text{ErrorMessages}$. The selection of the first applicable method yields

void m(I_DataLogging DataLogging, I_ErrorMessages ErrorMessages).

Therefore, a mapping $f: \{\text{DataLogging}, \text{ErrorMessages}\} \rightarrow \{\text{DataLogging}, \text{ErrorMessages}\}$ is generated with $f(\text{DataLogging}) = \text{DataLogging}$ from *Required(LogServices)* and

$f(\text{ErrorMessages}) = \text{ErrorMessages}$ from *Required(LogServices)*.

³⁰ g maps *LoggingOptions* of *LogServices* to *LoggingOptions* of *Logging*.

If ambiguity should not be resolved arbitrarily, steps two and four of the algorithm from Section 4.11.1.1 have to be refined. Before presenting these refined steps, we discuss how far and why our algorithm differs from the handling of applicable methods in Java in cases in which more than one applicable method exists.

If more than one method declaration is applicable to a method invocation, the Java programming language [GJS] uses the rule that the most specific method is chosen. One method declaration is more specific than another, if any invocation handled by the first method could be passed on to the other one without a compile-time type error. In our context, a method declaration

$$\text{void } m(I_{R_{i_1}} R_{i_1}, I_{R_{i_2}} R_{i_2}, \dots, I_{R_{i_n}} R_{i_n})$$

is more specific than a method declaration

$$\text{void } m(I_{R_{j_1}} R_{j_1}, I_{R_{j_2}} R_{j_2}, \dots, I_{R_{j_n}} R_{j_n})$$

if $\forall k, 1 \leq k \leq n : I_{R_{i_k}} \preceq I_{R_{j_k}}$. A method is said to be maximally specific for a method invocation, if it is applicable and there is no other applicable method that is

³⁰ f and g are the mappings between corresponding sets of services of two complementary plugs as introduced in term 4.4.7. g maps the names of the provided services of the plug *LogServices* to the names of the required services of the plug *Logging*.

more specific. If there is exactly one maximally specific method, then it is in fact the most specific method; it is necessarily more specific than any other method that is applicable. It is possible that no method is the most specific one, because there are two or more maximally specific methods.

Because our method declarations result from permuting the required services, no method can be more specific than another one. To show that this statement is true, assume the opposite. Without loss of generality let

$$\begin{aligned} 1) & \text{ void } m(I_R_1 R_1, I_R_2 R_2, \dots, I_R_n R_n) \quad \text{and} \\ 2) & \text{ void } m(I_R_{i_1} R_{i_1}, I_R_{i_2} R_{i_2}, \dots, I_R_{i_n} R_{i_n}) \end{aligned}$$

be two method declarations with different signatures generated by our algorithm, where the second declaration is more specific than the first one. Let $h : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ be the bijective mapping corresponding to the permutation of the required services with $h(j) = i_j$. Thus, the j -th parameter of the first method is R_j and the corresponding parameter of the second method is $R_{h(j)}$. As the second method is more specific than the first one, for all $j \in \{1, \dots, n\}$: $I_R_j \succeq I_R_{h(j)}$. As the signatures of both methods differ, there exists $k \in \{1, \dots, n\}$ such that $I_R_k \neq I_R_{h(k)}$. As h represents a permutation, there exists $l \in \{1, \dots, n\}$ such that $h^l(k) = k$, where $h^2(k) = h(h(k))$ and $h^l(k) = h^{l-1}(h(k))$. As the second method is more specific than the first one, it follows that $I_R_k \succeq I_R_{h(k)} \succeq I_R_{h(h(k))} \dots \succeq I_R_{h^l(k)} = I_R_k$ and thus $I_R_k = I_R_{h(k)}$. This contradicts $I_R_k \neq I_R_{h(k)}$.

Therefore, as no method can be more specific than another one, every applicable method is already a maximally specific method.

Now, the refined steps of the algorithm are presented:

Step two: This step differs from step two of the original algorithm mainly in the handling of permutations when permutations generate method signatures for which the types of all parameters are identical to the types of the parameters of an already existing method declaration. For every generated method declaration d there exists additionally a set N_d of n -tuples $(R_{j_1}, R_{j_2}, \dots, R_{j_n})$ representing all sequences of required service names leading to the same sequence of parameter types as the method declaration d . If this set contains more than one n -tuple, the method declaration d will lead to an ambiguity, if d is applicable.

Step four: This step differs from step four of the original algorithm in the handling of applicable methods.

If no applicable method can be found, the algorithm stops with an error. Otherwise let D be the set of all method declarations which are applicable and

$$M = \bigcup_{d \in D} N_d \text{ with } N_d \text{ as declared in step two.}$$

If $M = \{(R_{j_1}, R_{j_2}, \dots, R_{j_n})\}$, then the mapping $f : \{P_1, \dots, P_n\} \rightarrow \{R_1, \dots, R_n\}$ is unambiguous: $f(P_i) = R_{j_i} \forall i, 1 \leq i \leq n$.

If $|M| > 1$, a mapping would be ambiguous. In this case, the set M is offered to the user to select a n -tuple to be used for the mapping.

As already discussed above, this ambiguity can not be resolved automatically by looking for a most specific method as in Java, since in our context every applicable method is already a maximally specific method.

Example 4.11.3 (Ambiguous Mapping) *In example 4.11.1 part b) the following method declarations were already generated in step two:*

d_1 : *void m (I_DataLogging DataLogging, I_ErrorMessages ErrorMessage)*

d_2 : *void m (I_ErrorMessages ErrorMessage, I_DataLogging DataLogging).*

Using the refined step two we must additionally determine the sets N_{d_1} and N_{d_2} .

These are: $N_{d_1} = \{(DataLogging, ErrorMessage)\}$ and

$N_{d_2} = \{(ErrorMessage, DataLogging)\}.$

As already shown in step four of example 4.11.1 part b) both method declarations are applicable to the sequence of provided services (DataLogging, ErrorMessage) belonging to the plug Logging of Client 2. These methods are also maximally specific. Using the refined step four from above we have to determine the set

$$M = \bigcup_{d \in D} N_d, \text{ where } D = \{d_1, d_2\}.$$

This yields: $M = \{(DataLogging, ErrorMessage), (ErrorMessage, DataLogging)\}.$

Since $|M| = 2$, the set M is offered to the user to select a tuple of his choice e.g.

(ErrorMessage, DataLogging). After the selection the following mapping will be generated:

*$f: \{DataLogging, ErrorMessage\} \rightarrow \{DataLogging, ErrorMessage\}$ with
 $f(DataLogging) = ErrorMessage$ and $f(ErrorMessage) = DataLogging.$*

4.11.2 Composing Plugs when Creating Composite UCM-Components

When creating composite UCM-components, every plug Pl belonging to the component interface CIN has either to be linked to a fitting plug of one internal part or it can be composed of smaller plugs and single services of several internal parts. In the following we describe an algorithm for composing plugs.

One starts with plugs of parts. The conditions to hold for a plug Pl' of a part pc typed by CI_{part} which should be linked to the greater plug Pl of CIN are:

1. $\exists Pl'' = (PS', RS')$ with $PS' \subseteq Provided(CIN, Pl) \wedge RS' \subseteq Required(CIN, Pl) :$
 $PLtype(CI_{part}, Pl') \preceq PLtype(CIN, Pl'').$
2. $\forall S \in PS' \wedge \forall S \in RS' : S$ is not already linked to any other internal entity.

For a plug Pl' with

$$(*) \quad |Provided(CI_{part}, Pl')| > |Provided(CIN, Pl)| \quad \text{or} \\ |Required(CI_{part}, Pl')| > |Required(CIN, Pl)|$$

it is impossible to find a 'subplug' Pl'' of Pl so that $PLtype(CI_{part}, Pl') \preceq PLtype(CIN, Pl'')$. Therefore, such plugs are deleted from a list L of plugs which may be possibly linked to 'subplugs' Pl'' of Pl . From the remaining list of plugs created by the tool the user selects the one (Pl_{sel}) he wants to be linked to Pl . The tool may support the link by the algorithm described below. If the link is not successful, Pl_{sel} is deleted from L and the user may proceed by selecting another plug from L or by aborting this activity. If the link is successful, the services of Pl are reduced by the services linked to services of Pl_{sel} . Pl_{sel} is deleted from L as well as all other plugs which no longer satisfy condition (*). From this new list the user selects a further plug and so forth until the user aborts this activity. He may proceed by linking fitting stand-alone services of parts to the remaining unlinked services of Pl , if any.

If the user selected a plug Pl_{sel} of a part pc' of type CI'_{part} to be linked to a 'subplug' of Pl , the tool may support the linking by determining a fitting 'subplug' Pl'' of Pl which may be linked to Pl_{sel} . The determination of Pl'' is done as follows.

Let $n_{P_{sel}} = |Provided(CI'_{part}, Pl_{sel})|$ and $n_{R_{sel}} = |Required(CI'_{part}, Pl_{sel})|$.

$$\text{Let } PS = \{PS' | PS' \subseteq Provided(CIN, Pl) \text{ with } |PS'| = n_{P_{sel}}\} \quad \text{and} \\ RS = \{RS' | RS' \subseteq Required(CIN, Pl) \text{ with } |RS'| = n_{R_{sel}}\}$$

Select $Pl'' = (PS', RS')$ for arbitrary combinations of $PS' \in PS$ and $RS' \in RS$. Check, whether $PLtype(CI'_{part}, Pl_{sel}) \preceq PLtype(CIN, Pl'')$. If OK, then a link may be established between fitting services of Pl_{sel} and Pl'' and the search can be abandoned. If not OK, further combinations have to be checked.

The algorithm may be improved by restricting the sets PS and RS to

$$\{PS' | PS' \subseteq Prov(CIN, Pl)\} \quad \text{and} \quad \{RS' | RS' \subseteq Req(CIN, Pl)\}$$

whereas $Prov(CIN, Pl) \subseteq Provided(CIN, Pl)$ and $Req(CIN, Pl) \subseteq Required(CIN, Pl)$ are determined by excluding services which do not fit to any of the provided resp. required services of Pl_{sel} . That is,

$$Prov(CIN, Pl) = \bigcup_{P' \in Provided(CI'_{part}, Pl_{sel})} \{P | P \in Provided(CIN, Pl) \wedge PStype(CI'_{part}, P') \preceq PStype(CIN, P)\}$$

$Req(CIN, Pl)$ is declared accordingly.

The process of composing Pl from smaller plugs and stand-alone services of internal constituents is completed, if all services belonging to Pl are linked to internal constituents.

Chapter 5

Evaluation

This chapter evaluates the concepts from Chapter 4 and introduces some additional features supporting visual composition. Section 5.1 presents the *Bean Plug Composition Environment* or short *BPCE*, a tool to compose and test JavaBeans which can be connected via events, services or plugs visually. Service and plug connections are available to beans conforming to our component model and all other beans enriched with suitable specification information describing the services and plugs provided and/or required by the bean. This tool demonstrates how JavaBeans can be used as atomic components and how the support for visual composition can look like. Section 5.2 presents the *Composite Component-Builder*, a tool which allows us to compose UCM-components hierarchically. It demonstrates that arbitrary composite UCM-components of an arbitrary level of complexity can be built and their methods called without knowing their services and service interface types in advance.

5.1 The BPCE as a “Proof of Concept”

The *Bean Plug Composition Environment* or short *BPCE* is our extension of Sun’s BeanBox. The original BeanBox is a test tool for JavaBeans. It allows one to select JavaBeans from a tool box, to place corresponding instances onto a composition window, to configure them by changing their properties, and to connect bean instances based on events. The BPCE extends the features of the BeanBox. It supports our component model using JavaBeans as atomic components. Interconnections between component instances are extended to additionally support interface and plug connections. Interface and plug connections can be performed visually and are available to beans conforming to our component model and all other beans enriched with specification information describing its component interface (Section 4.1.2). Existing JavaBeans without specification information can still be composed in the well-known manner based on events. The tool analyzes the specification information provided by the components to know about their capabilities and requirements and provides advanced support in connecting component instances and checking for consistency.

All component instances needed for an assembled application are selected from a tool box and placed onto a composition window. Two component instances residing in the composition window are connected via interfaces by first selecting a source and one of its required services and afterwards selecting a target and one of the target's provided services. After the selection is finished, the tool connects both component instances by internally calling the default connect-method of the selected required service of the source. A similar process holds for plugs. The selected component instances are referred to as "source" and "target" according to the direction of the selection process.

Every component instance in the composition window is represented by a wrapper object which holds references to the corresponding instance of a JavaBean realizing the behavior of the component, a component interface object created from the component interface description, and a connection info object which stores information on the connections already established for the required services of the component instance.

In the following, the assembly process is described in more detail referring to some screen shots of the BPCE.

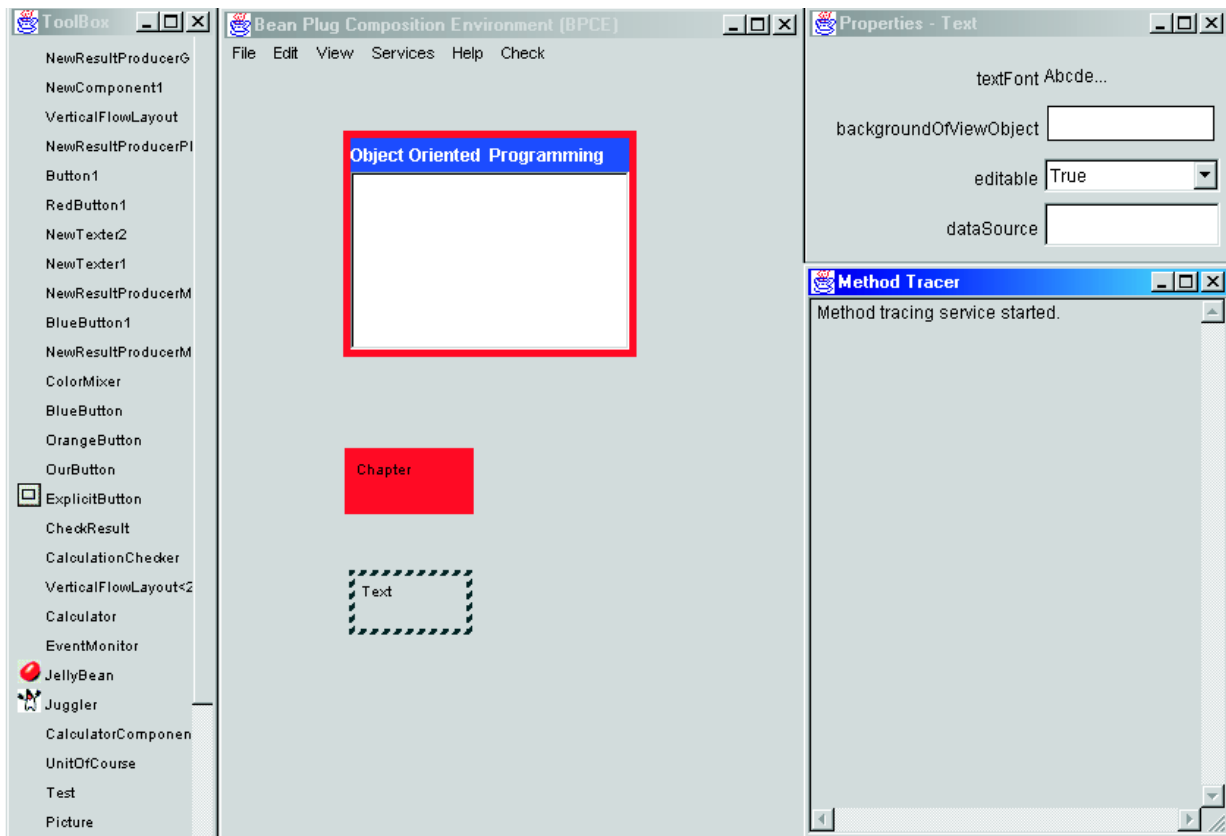


Figure 5.1: User Interface of the BPCE

Figure 5.1 shows the user interface of the BPCE which is made up of four windows: the tool box on the left, the composition window in the center, the property editor window on the right, and below a window showing some tracing information. The tool box

is filled with components from the component pool (Section 4.5) realized by a directory with a predefined name. The property editor is used to set properties of the active component instance in the composition window as e.g. for- and background colors or text fonts.

The composition window in Figure 5.1 already contains three component instances. The top most component instance labelled *Object Oriented Programming* (in the following denoted by source) should be connected to the one labelled *Chapter* (in the following denoted by target) via interfaces. To be able to determine the required services of the source, the source has to be selected by a mouse click to become the active component instance. To get its list of required services, the menu item *interface* from the *edit* menu has to be selected. Then the required services of the source component are listed in a popup menu. Every required service is marked as optional or mandatory depending on its limits on the number of connections and the connections already established. The required service which should be connected to a provided service of the target has to be selected from this popup menu (see left figure in Table 5.1).

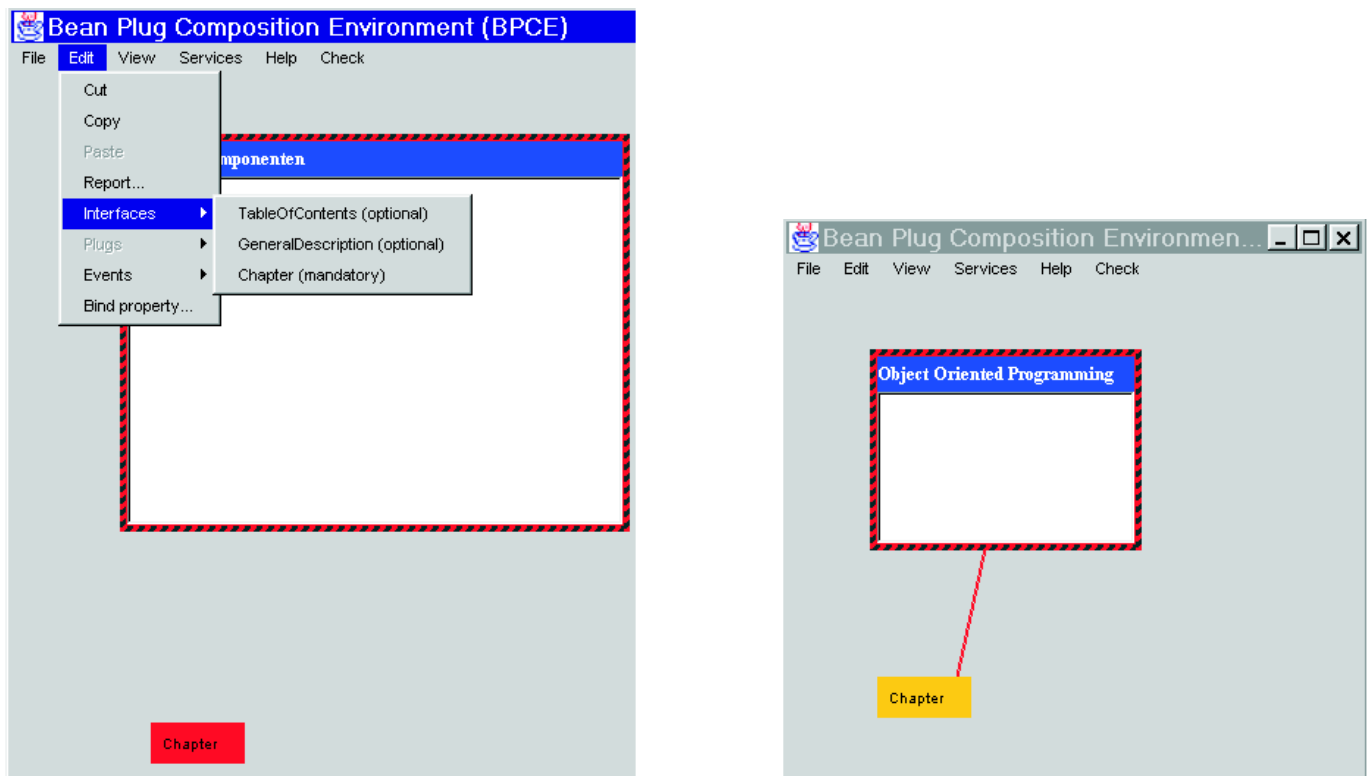


Table 5.1: List of the Required Services of the Selected Source Component (left figure) Selection of a Service Provider, here *Chapter* (right figure)

To create the list of required services (term 4.1.4) used to fill the popup menu, the BPCE asks the component interface object for all required services declared in the component interface not belonging to a plug (term 4.1.15) including the information concerning their lower and upper limit on the number of connections (Section 4.1.1.3). For every required service R obtained, the BPCE asks the connection info object for the number of already established connections. If this number exceeds the lower limit min_R , R is marked as optional even if R is declared to be a mandatory required service (term 4.1.6) in the component interface. This is due to the fact that the needed number of connections is already reached and thus further connections may be done, but are not required. If the number of connections already established equals the upper limit max_R , then R is not added to the list of required services. Thus, a user can no longer select this required service for a connection. Thereby the BPCE ensures that the number of connections established does not exceed max_R .

After the selection of the required service of the source which has to be connected to a provided service of the target, the target (here *Chapter*) has to be selected by a mouse click to become the active component instance. The process of selecting a target is visualized by dragging a line from the source to the target (see right figure in Table 5.1).

After a target was selected, a window opens which shows all provided services of the target matching the selected required service of the source (see Figure 5.2). From this window the provided service to be connected to the previously selected required service of the source component instance has to be chosen.

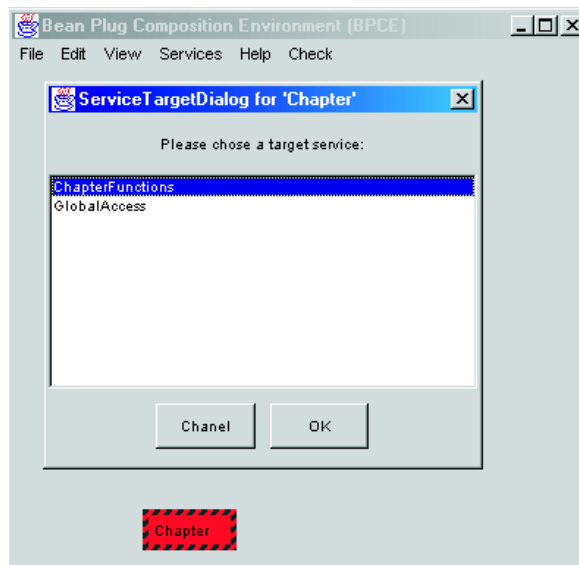


Figure 5.2: List of Matching Provided Services

To create the list of provided services shown in the window, the BPCE asks the component interface object for all provided services (term 4.1.3) declared in the component interface not belonging to a plug. For every provided service P obtained, the BPCE

checks, whether its corresponding service interface type is a subtype of the service interface type of the selected required service R . If true, P and R are matching services (term 4.4.5) and can thus be interconnected (theorem 4.4.6). Therefore, P is added to the list. If P and R are no matching services, P is not added to the list. Thus the list only contains provided services fitting to the selected required service. Thereby the BPCE prevents invalid connections.

After a provided service was selected from the list of matching services, the BPCE establishes the connection. To establish the connection, the BPCE asks the bean instance realizing the source component instance for the connection point object CPO_R (term 4.1.11) belonging to R via the bean’s service access interface (term 4.1.13). Similarly, the BPCE asks the bean instance realizing the target component instance for the service object SO_P (term 4.1.8) bound to P . It then calls the default connect-method obtained from the component interface object on CPO_R with SO_P as the actual parameter for the formal parameter representing the service provider.

Similarly, connections between component instances via plugs may be established. Instead of services, plugs are listed as sets of required and provided services.

The process of connecting two component instances described so far is additionally supported by several other features simplifying the composition process significantly. In the following, the features already presented so far as well as some new features are summarized for services, but they also hold for plugs. In the following description, *min-connections* denotes the lower limit, *max-connections* the upper limit on the number of connections corresponding to a required service.

Edit/Interfaces (Table 5.1) The list of the required services of the source is reduced to those services which are not yet fully connected. The list also contains information on whether a required service is optional or mandatory. The status changes from mandatory to optional, as soon as all needed connections (*min-connections*) are established.

A required service is called *fully connected*, if already *max-connections* component instances are connected to it.

(How this feature is implemented is already described on page 226.)

ServiceTargetDialog (Figure 5.2) The list of provided services of the target is reduced to those services matching the required service of the source.

(How this feature is implemented is already described on page 226.)

Mark components still to be connected (Figure 5.3) All beans in the composition window which still have open mandatory required services are marked by a red rectangle surrounding them. *Open mandatory required service* means, that the lower limit on the number of connections is not yet reached.

(The BPCE implements this feature by checking all component instances in the composition window. For each of these component instances C and each of its required services

R the BPCE asks the connection info object corresponding to C for the number of already established connections for R. If this number is less than the minimum number of connections for R, C has open mandatory required services and must be marked.)

Mark fitting target components (Figure 5.3) The BPCE provides a connection support by marking all component instances in the composition window, which would be suitable service providers (term 4.1.5) to the selected required service of the source going to be connected.

In this context the term *suitable service provider* is used more restrictive as in term 4.1.5 as we additionally demand that the service provider is not excluded by the constraint explained in Section 4.1.1.3 paragraph *Different Service Providers*.

(For the following explanations, we call the selected required service of the source going to be connected R. The BPCE implements this feature by checking all component instances in the composition window. For each of these component instances C and each of its provided services P the BPCE checks, whether P matches R. If P matches R, the BPCE then checks additionally, whether C is not excluded by a constraint of kind Different Service Providers (term 4.1.20). Thus, if R belongs to a constraint set of a constraint of kind Different Service Providers, the BPCE checks, whether a provided service of C is already connected to one of the other required services belonging to the constraint set by asking the connection info object of C for the necessary information. If so, C is no longer a suitable service provider.)

Guide connection (Figure 5.3) An automated connection support can be chosen. If selected, the BPCE behaves as follows: If a target is to be connected to a source and the target has still open mandatory required services, the BPCE gives an error message. The user may then choose other component instances and connect them to the target first. Not until all mandatory required services of the target are at least connected to *min-connections* service providers, the target is in turn connected to the source.

The last three options can be enabled or disabled using the *Check* menu as shown in Figure 5.3.

After all component instances are selected and wired together, the user may save the assembly by selecting the menu item *Save* from the *File* menu.

Before the assembly is actually saved, the assembly is checked for consistency (Section 4.4). If any of the component instances in the composition window has still open, mandatory required services, the BPCE gives an error message and rejects to save the assembly. If there are still optional required services which are not connected to any service provider, a warning is given, but the assembly is saved. Existing interconnections do not have to be checked for consistency, because by the way the tool allows one to establish connections it enforces proper connections.

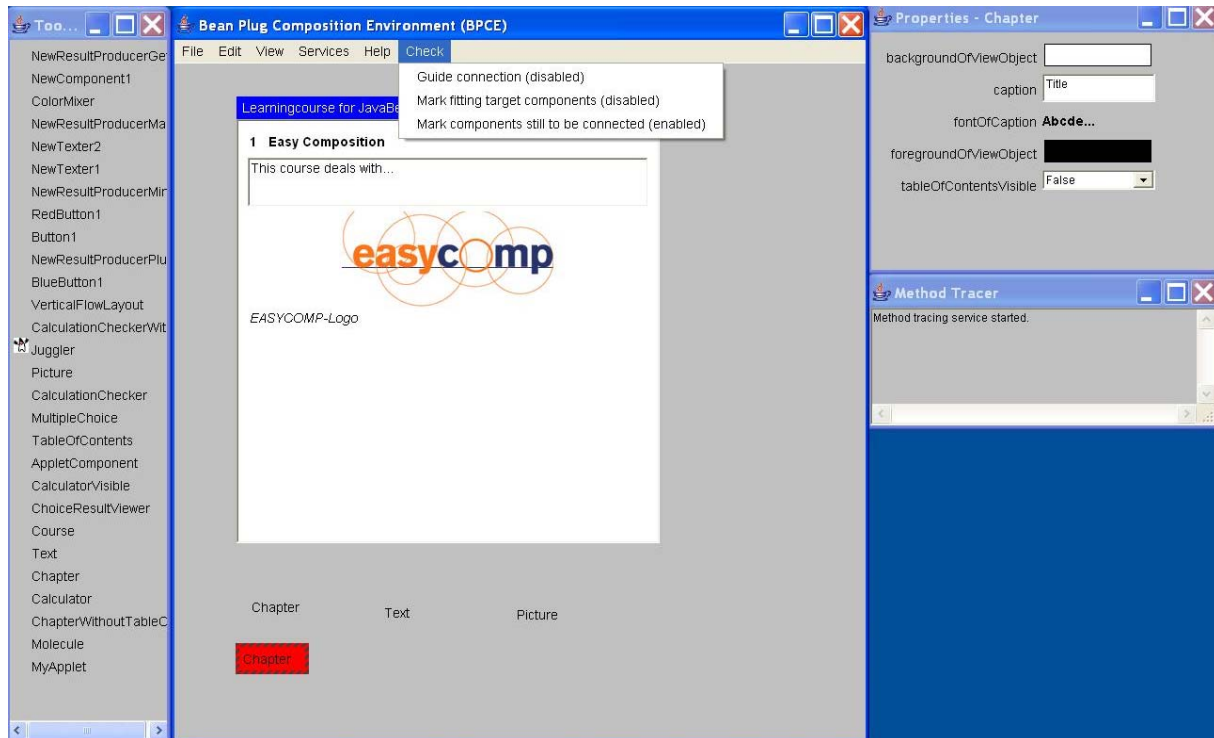


Figure 5.3: Enabling or Disabling additional Connection Support

5.2 The CC-BUILDER

We developed a tool called *Composite Component-Builder* (CC-BUILDER) used to create new applications and new composite UCM-components from atomic and other composite UCM-components. Thus, the CC-BUILDER allows composite UCM-components to be built hierarchically.

Hierarchical Composition: The typical composition process to build new components from existing ones looks as follows.

A programmer selects components from the list of available components and places instances of them onto the composition window. Then he selects each of the component instances C residing in the composition window which has at least one required service. He selects one of the mandatory required services R of C by clicking on the representation of the service, a little button labelled 'REQ' on the right hand side of the component. The service name appears as soon as the mouse is moved over the button. Then he selects a fitting provided service of another component instance in the composition window by clicking on its button representation labelled 'PROV'. The connection is visualized by a line drawn between both services. The programmer continues this process, until all mandatory requirements are resolved except for those which are to be exposed by the new composite component just being built.

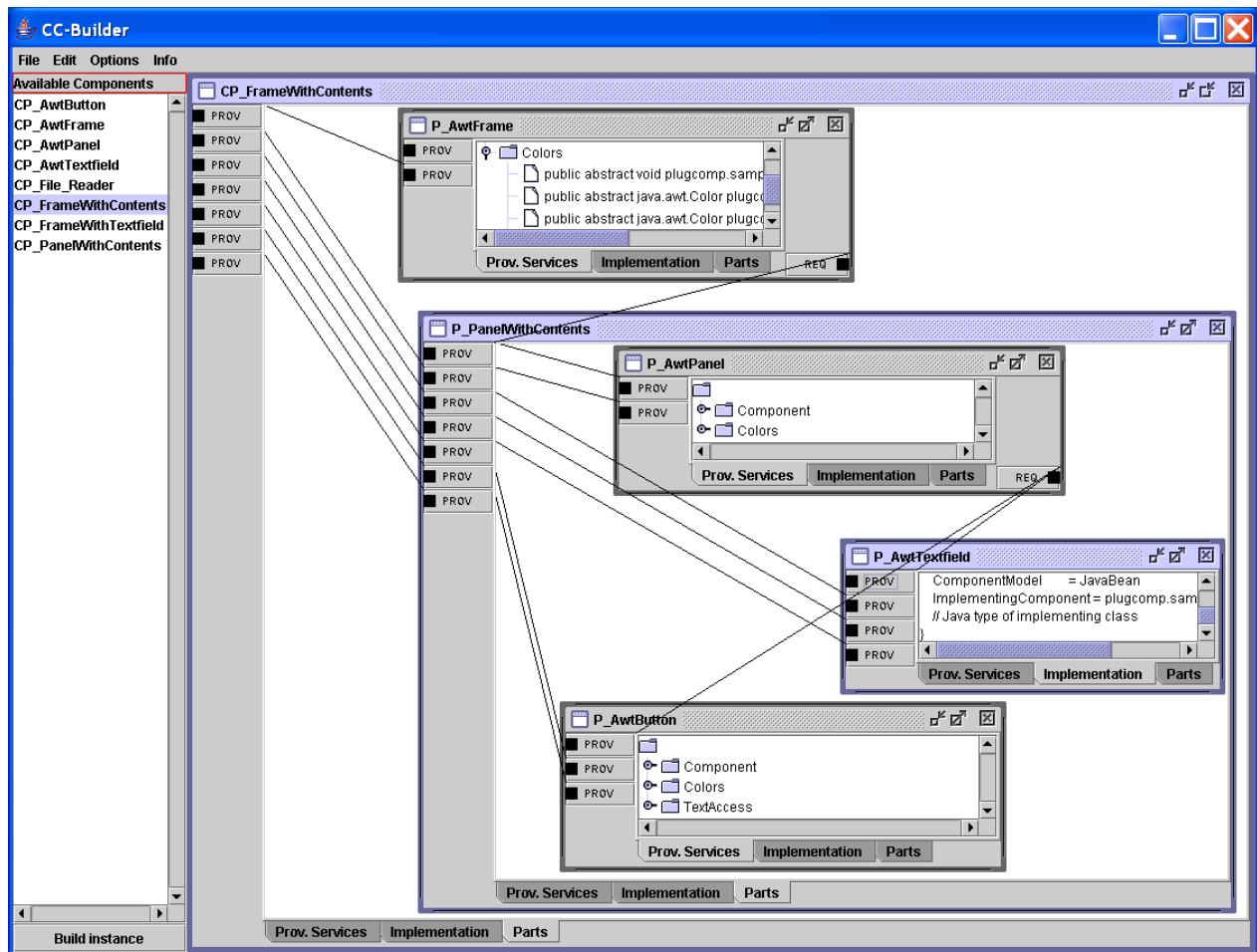


Figure 5.4: Hierarchically Composed Components

All optional required services the programmer wants to connect are connected using the same process. The programmer can save the new composition as a new component. He can select between the following two possibilities:

1. The tool creates a fitting component interface declaration for the new composite component automatically by a predefined algorithm.
2. The programmer selects each service he wants to be exposed explicitly and, if necessary or desired, declares a name for the service in the context of the new composite component. The tool then generates the corresponding service for the composite component and visualizes the export by a line drawn from this service to the service of the component instance exposing it.

The newly created component can further be composed to a higher level component. Figure 5.4 shows an instance of a composite component named “P_PanelWithContents”

which consists of three parts and which is itself used to build a higher level component “CP_FrameWithContents”.

Creation of a New Component Interface: If the tool creates a fitting component interface declaration for the new composite component automatically, it uses the following predefined algorithm: All services of constituents not connected to services of other constituents are exported. For every exported service the tool has to declare a service in the component interface which gets the same name and service interface type as the exported one. For required services the tool adopts the connection point and cardinality types, too. All already connected required services for which the lower limit on the number of connections is not yet reached are exported, too. When exporting such services, the lower and upper limits on the number of connections are reduced by the number of already established connections. If two services of the same kind (provided or required) having the same name are exported, the names of the two corresponding services of the component interface are built by appending a number to the original name. This is necessary since services of the same kind belonging to the same component interface must not have the same names.

Components and the User Interface: The CC-Builder uses a component pool realized by a directory which is made known to the CC-Builder at program start up. All atomic and composite UCM-components belonging to this component pool are listed in the left most panel of the CC-Builder. From this list a component can be selected for composition. Instantiation of the selected component is done by a mouse click on *Build instance*. The instance generated is represented by a window shown in the composition window of the CC-Builder. Each component instance has three different views: a *functional view*, an *implementation view*, and a *part view*.

For every newly created component instance the functional view is shown by default. It shows all provided services of the component instance as folders. Each folder can be opened which results in showing all methods belonging to the service (see the component instance labelled “P_AWTFrame” in Figure 5.4). A method can be executed by double-clicking on it. The user has to enter a value for every of the method’s arguments. In this first version only arguments of primitive data types are supported and of types having a nullary constructor that is, a constructor without parameters. If the instantiation of a component causes a visible GUI-element to be shown as e.g. a frame, execution of methods can often be observed directly. If a method call changes the appearance of the shown GUI, e.g. its foreground or background color, this will be directly visible.

The other views can be shown by selecting the corresponding riders labelled “Implementation” and “part” at the bottom of the window representing the component instance. The implementation view shows the UCM-code of an atomic or composite component as shown by the component instance labelled “P_AWTTextField” in Figure 5.4. The part view shows the parts a composite UCM-component consists of, the existing links to services of its component interface as well as the interconnections between its parts. The component instance labelled “P_PanelWithContents” is represented as a part view.

Each component instance represents the services it provides by buttons labelled “PROV” on the left hand side of the component instance. Similarly, required services are represented by buttons labelled “REQ” on the right hand side of the component instance. The name of the service represented by a button is shown as a tooltip as soon as the mouse is moved over the button.

Look & Feel: A user may select between different appearances of the GUI elements. Thus, a user may select a look & feel according to his needs. The following screen shots show the selection of a style as well as the representation of the user interface in Motif- and Windows-style.

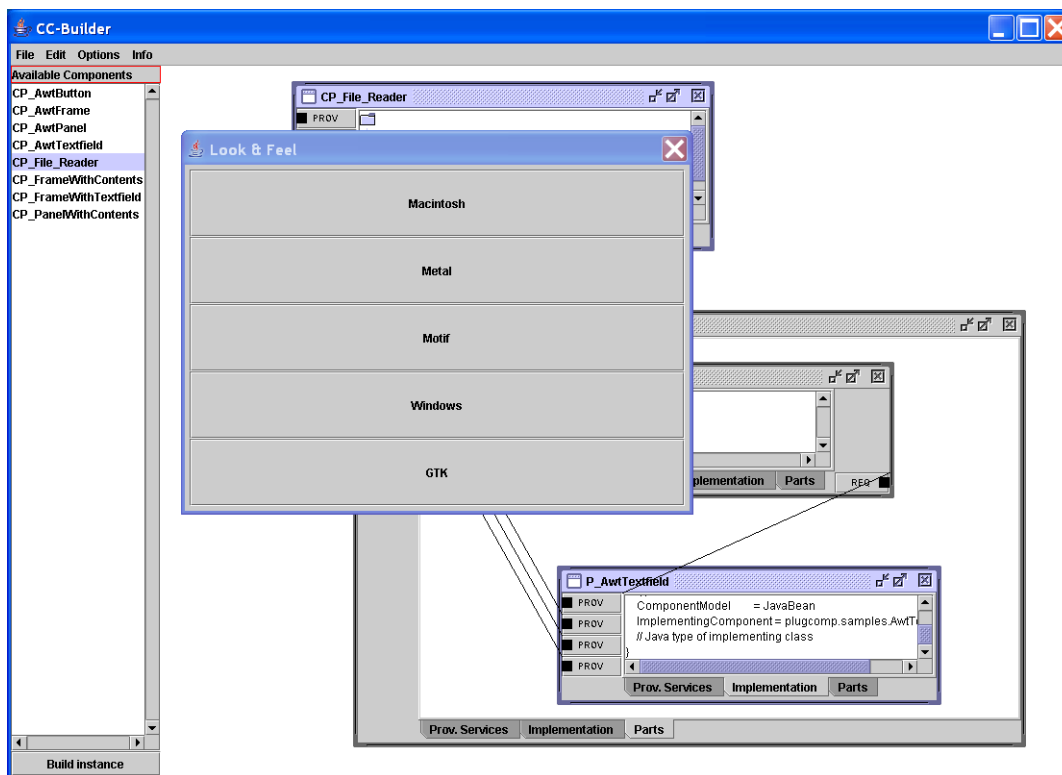


Figure 5.5: Selection of a Style

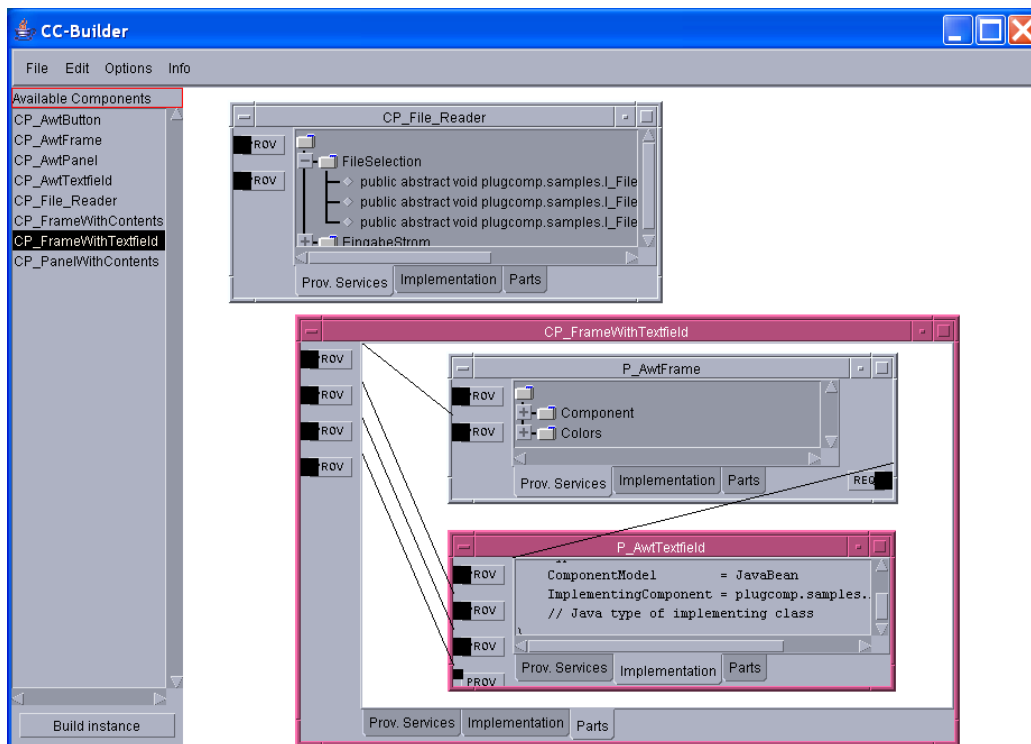


Figure 5.6: Motif-Style

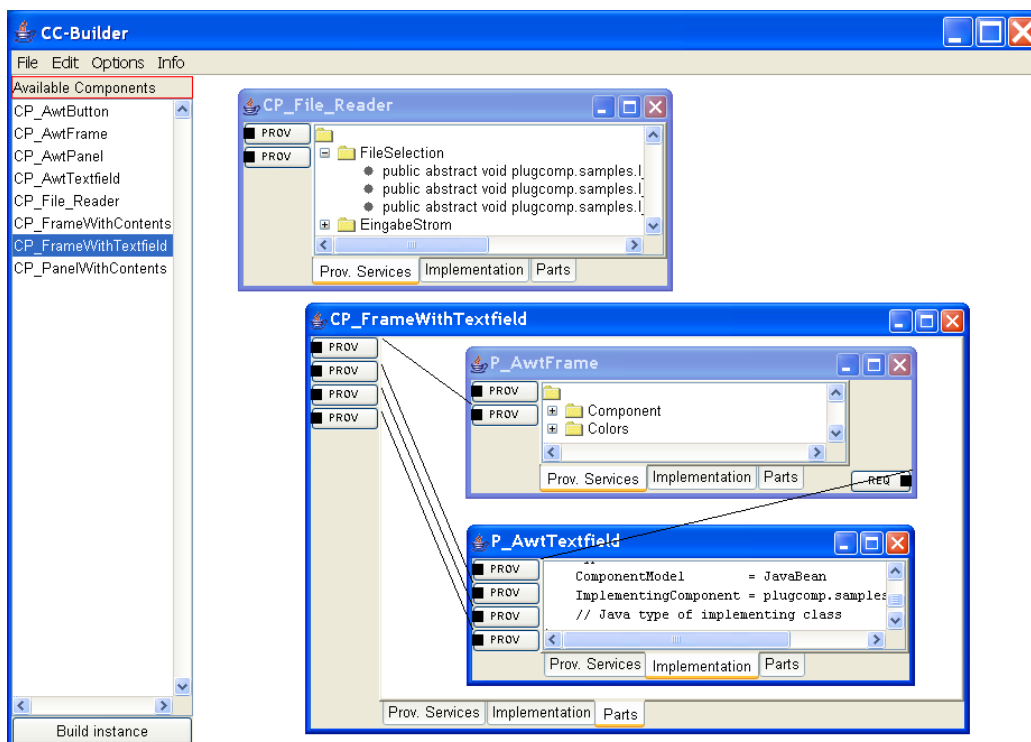


Figure 5.7: Windows-Style

Chapter 6

Related Work

This chapter presents various scientific fields which deal with component based development. We shall focus on what constitutes a component, how components can be built, which kinds of composition are supported, which dependencies between components can be modelled, whether it is possible to build new components from existing ones, and whether and how types and subtypes are supported. The first two fields concern language support for component based development, the other two fields concern system design.

In the area of language support, we discuss component-oriented languages and composition languages. Whereas component-oriented languages try to integrate component development and component composition, composition languages are not targeted to develop new components from scratch. Instead, their focus is on composing applications and, in some approaches, new components from already existing ones delivered by third parties. They are related to our description of composite UCM-components.

In the area of system design, we deal with architecture description languages (ADLs) and the UML 2.0-approach to component based development. ADLs provide language support to model systems built from interconnected components in the context of software architecture. Here, means to represent component and system hierarchies are essential. UML 2.0 is related to our approach by its graphical means to denote interconnected components as well as component hierarchies.

6.1 Component-Oriented Languages

Programming languages contribute to component based development by providing programming language constructs dealing with components as first class entities like classes (see e.g. [SC00a, SC02, Sre01, ACN01, ACN02, Zen02]). As an important feature, such languages provide explicit language support for composition. Existing component oriented programming languages differ significantly from one another. The description of all existing approaches would go beyond the scope of this thesis. Thus only some approaches are discussed here. Component Pascal [Com01a] is mentioned for the sake of

completeness although it is not a component oriented language in our sense as it does not treat components as first class entities. It does not provide extra communication paths between components nor composition concepts which allow one to simply stick components together or to nest component instances inside others.

Our description of the languages discussed is structured into three main parts concerning components and their interconnections, component hierarchies, as well as subtyping and dependencies.

6.1.1 ArchJava

ArchJava [ACN01, ACN02, Arc] is a component-oriented language which tries to smoothly integrate architectural aspects of a software system into a programming language thereby focusing on components and their communication paths.

Components and Interconnections: Components are declared by the keyword *component class*. The declaration of a component comprises the specification of the component's interface as well as its implementation. In contrast to our approach, there is no possibility to define a component interface apart from its implementation such that the interface can be implemented by different component classes. Components have ports to communicate with each other. A port is a named entity declaring all methods involved in a communication via this port. Ports can contain provided as well as required and broadcast methods. Component instances at the same level are only allowed to communicate via their connected ports. Provided methods of a port must not be called directly, only via an established connection. In contrast to required methods which can only be bound to one fitting provided method, a broadcast method can be bound/-connected to several fitting provided methods. Calling a broadcast method results in calling all the connected provided methods, one after the other.

A connection is established by an explicit connect-statement which designates all ports involved in the connection. A port P of one component instance can be connected to **several** other ports of other component instances simultaneously through one connection as far as these ports collectively provide all the methods required by P and P as well as the other ports can fulfill all the requirements of any of the other ports. For every required method belonging to one of the ports exactly one provided method with the same signature may be provided by the other ports which is then bound to the required method. As more than two ports can be involved in the same connection, it is especially possible that P is connected to two ports A and B , where A provides all the methods P requires and B requires all the methods P provides. Thus P does **not enforce** a bi-directional connection between itself and a single other party. Although bi-directional connections on method level can be realized through ports, a port is, in contrast to our plugs, not primarily intended to model bi-directional connections. Instead, a port simply acts as a communication channel between two or more component instances listing all methods involved in this kind of communication between all participants. A concept

like our plug, grouping ports to higher level entities, does not exist. In contrast to our approach, connections once established can not explicitly be removed.

Component Hierarchies: Component instances can be contained in other component instances, their container. A container can communicate with its parts via ports or by directly calling the public methods of the parts not belonging to a port. Port communication can be achieved by connecting a private port of a container to a public port of one of its parts. As containers can directly call the public methods of its parts, communication between component instances is not strictly interface-based. In our approach all interaction is done through our services. Additionally, component instances contained in our composites can be declared by their component interface types only which allows different implementations to be chosen without affecting the composite components.

Subtyping and Dependencies: In ArchJava, a component class can extend another component class. Inheritance of component classes is similar to inheritance in Java. Fields, methods, ports and connections are inherited from the component superclass. New fields, methods and ports can be added as well as new connections between new fields of component type. Inherited ports are immutable with respect to the method signatures and the port modifiers (provides, requires). Especially, provided or required methods must not be added to an inherited port. But as in Java, provided methods can be overridden that is, their implementation can be changed while their signature remains unchanged. New ports must not contain new required methods. We allow additional provided methods in a subtype as well as additional required services as long as they are optional. Allowing inheritance between components, ArchJava supports white box reuse for components. Thus in ArchJava the fragile base class problem [MS97] holds for components as it does for normal Java classes. We only allow black box reuse of components, the recommended kind of reuse in componentware, to avoid the problems known from the white box reuse of classes.

In ArchJava, dependencies on other components are declared in terms of required methods only, not in terms of entire interfaces. Although a port can group several required methods, the corresponding provided methods they are connected to need not be provided by one component only. Instead the corresponding provided methods can belong to several different components. In ArchJava there is no means to declare, whether a dependency is optional or mandatory.

Miscellaneous: As components are mapped to Java classes and interfaces comprising the required methods of their ports at compilation time, components will not really be deployment units or units of third party composition which can be used as a black box. Industrial component models can not be integrated into ArchJava.

6.1.2 ComponentJ

Components and Interconnections: ComponentJ components [SC00a, SC02] communicate to the outer world only through their component interfaces which are defined in terms of provided and required ports. A port has a name and is typed by a Java interface declaring the methods belonging to the port. The functionality of a provided port is implemented by so-called method-blocks inside the component or by provided services of internal parts (see Component Hierarchies). A method-block is a named block enclosed in curly brackets which contains method declarations and implementations. A method-block must be attached explicitly to a provided port by a plug-statement¹. Although not explicitly stated, we assume that the attachment is only valid, if at least all methods declared by the port type are implemented in the method-block.

A method belonging to a port can directly be called by a client of the component. In addition, provided ports of one component can be connected to required ports of other components to fulfill their requirements. A connection between a required and a provided port of two components is established by the same plug-statement already used to attach method-blocks to ports. A connection can only take place, if the type of the provided port is equal to or a subtype of the type of the required port.

In contrast to our approach, connections once established can not be released. This can be a disadvantage if components should be assembled using an assembly tool. Such a tool creates instances of components selected by a user and connects them according to the user's needs. If the user wishes to reconfigure the assembly, it may become necessary to release existing connections. In addition, it is not possible to declare optional dependencies. Thus no support exists for possible extensions or notification services. Higher level entities on top of ports as our plugs, enforcing and simplifying bi-directional connections between components, are not available.

Components in ComponentJ, like components in ArchJava, specify their interface to the outer world implicitly by declaring their ports in the context of their implementation. That is, similar to classes in Java, components in ComponentJ have an implicit interface which means that the interface is not declared explicitly by a separate language construct, but has to be inferred from the implementation. It is also possible to declare separate component interface types which can be compared to interface declarations in Java. Such component interface types consist of a set of provided and a set of required port-declarations only. But in contrast to Java and our approach, these component interface types can not explicitly be related to a component declaration via an implements relationship. Thus, component interface types are not introduced to declare an explicit relationship between component implementations and component interface types, but are introduced to type variables in a composite component thereby allowing to select between several different component implementations (see 'Component Hierarchies').

¹Plug-statements correspond to connect-statements in our approach.

Component Hierarchies: Component instances can contain other component instances as parts. This relationship is expressed in the outer component's code by some kind of variable declaration starting with the keyword *intro* followed by the name of the component to be used for instantiation followed by the name of the part. Instead of directly referring to the component to be used for instantiation, a component interface type can be used. A component implementation fitting to the component interface type can be assigned by a compose-expression or by parameter passing. Unfortunately, it is not explicitly mentioned, when an implementation fits to a component interface type.

A compose-expression creates a nameless component implementation at runtime and can directly be assigned to a variable of a component interface type. Parameter passing can be used, when components are return values of methods. This is e.g. useful for factory methods which generate component implementations by compose-expressions which depend on the actual implementations passed for one or more of the parts in the composite component. The following example from [SC02] shows port interface and component interface declarations as well as the declaration of components and the use of compose-expressions.

```
/* ----- */
port interface IPrint {
    void print(string);
}
component interface TPrint {
    provides IPrint p;
}
/* ----- */
port interface IDo {
    void doIt();
}
component interface TDo {
    provides IDo d;
}
/* ----- */
port interface IFactory {
    // Factory method returning a component of type 'TDo'
    TDo make(TPrint);
}

// Component declaration
component Factory {
    provides IFactory f;

    // Method block 'm' implementing method 'make' from
    // port 'f' of type 'IFactory'
    declare m {
        TDo make(TPrint printer) {
            // Compose expression creating a component which
            // provides a port 'd' of type 'IDo'. Thus, the
            // created component is of type 'TDo'. When creating
            // the component, the component 'printer' of type 'TPrint'
```

```

// which is passed to 'make' as parameter, is used as
// implementation for instantiating 'p'.
return compose {
    provides IDo d;
    intro TPrint p = printer;
    declare mm {
        void doIt() {
            p.p.print("Hello World");
        }
    }
    plug mm into d;
};
}
plug m into f;
}
/* ----- */

```

A provided port of an internal part can be used as the implementation for a provided port of the composite component through a connection between both ports. Similarly, a required port of the composite component can be connected to a required port of one of its parts. In accordance with our approach, an internal provided port implementing an external one must be of a subtype, an internal required port connected to an external required port must be of a supertype.

As long as compose-expressions are not used, parts in composite components have to be specified by referring to a special component implementation which can not be changed without affecting the code of the composite component. Variable declarations in our composite UCM-components can be made using component interface types only which allows different implementations to be chosen without affecting the composite components.

Subtyping and Dependencies: All dependencies of a component have to be resolved at component instantiation time. Thus, when a component instance with required ports is created, fitting service providers have to be specified as well as the required ports to which they will be plugged as e.g. in

```
tm = new CToDoExtension() {plug new DList() into list;}
from [SC00a].
```

Our component instances can be created even with open requirements. This allows several component instances to be loaded and configured in a visual environment and then connected to each other visually. In our approach, only complete applications are forbidden to contain components with mandatory requirements which are not satisfied inside the application. Whether a new component is built or an application, is determined by the visual builder tool through corresponding menu-entries selected by a user.

In ComponentJ a component type T_1 is a subtype of a component type T_2 , if T_1 has at least as many provided ports as T_2 and at most as many required ports as T_2 . In

addition, the types of provided ports of T_1 must be equal to or subtypes of the types of the corresponding provided ports of T_2 . A contravariant relationship holds for corresponding required ports. Although not explicitly stated, corresponding ports are determined based on equal port names which corresponds to our strong subtyping. Our weak subtyping is not supported. Allowing subtypes to have less requirements can cause problems in composite components having as part a component of the supertype. Required ports of the supertype no longer contained in the subtype and occurring in a plugging-operation can no longer be accessed. Due to this problem, our subtype definition does not allow a subtype to have less required services (ports) than its supertype as is allowed in ComponentJ. In addition, it allows additional optional requirements which enable e.g. additional notifications. As ComponentJ does not support entities like our plugs or constraints on interconnections, such entities can not be found in type and subtype definitions.

Miscellaneous: As in ArchJava, components are first mapped to Java classes and then compiled using a normal Java compiler. The resulting Java classes are packed into a JAR file for distribution. It is not clear whether these class files can be reused in ComponentJ code as being a ComponentJ component. However the class files can be used by pure Java code. Components can be instantiated and their ports accessed by special framework classes.

As in the case of ArchJava, ComponentJ does not support the integration of industrial component models.

6.1.3 Component Pascal

Component Pascal [Com01a, Com01b] is a programming language which combines imperative programming language features with object oriented features. It is a refinement of Oberon-2 [Mös98]. Although it calls itself a component-oriented language, it does not support components as first class entities. Instead, modules are called components as discussed below. Additionally, Component Pascal does not provide extra communication paths between components nor composition concepts which allow one to simply stick components together or to nest component instances inside others. Thus essential concepts of component oriented programming languages are missing.

Components and Interconnections: In Component Pascal, modules are called components. Modules are essentially the modules known from imperative programming languages. They differ however, in that they can contain classes and are runtime units. As runtime units, modules can be loaded into memory and run explicitly by executing one of their commands. In this context, a command is a parameterless procedure exported by the module. A command is specified by the module name, followed by the name of the procedure separated by a dot (e.g. `Trees.TestTree`, see the following example).

Modules can contain type, variable and procedure declarations as known from imperative programming languages. In addition, record types can be *extended*. The new type is compatible with the extended one and can contain additional field declarations. A record type extending another one denotes the name of the extended type in parentheses as in `CenterNode = RECORD(Node) ...END;`. Pointer types adapt the extension relationship of the record types they point to, thus `POINTER TO CenterNode` is an extension of `POINTER TO Node`.

Class declarations are based on record type declarations. There is no extra class-construct that is, classes are **not** declared as units comprising field and method declarations. Instead, class declarations essentially consist of a record type declaration and procedure declarations which are made apart from the type declaration. Procedures are correlated with the types they manipulate by an explicit 'this' or 'receiver'-parameter following the keyword `PROCEDURE` as in

```
PROCEDURE (t: Tree) Insert (node: Tree).
```

Here the 'this' resp. 'receiver' parameter is `t: Tree`. Procedures declaring a this-parameter are called *methods*.

If record types are classes that is, if for these types methods are declared operating on them, then a record type extension also comprises that additional methods can be declared operating on the extended type. This corresponds to subclasses providing new methods not known in their superclasses. Overriding of methods is also supported as shown in the following example. The code mainly combines code snippets from [Com01a]. It shows a class declaration (`Tree`) inside a module declaration². `CenterTree` is a subclass of `Tree` and overrides method `Insert` from class `Tree`.

```
MODULE Trees; (* exports: Tree,Node,CenterTree,CenterNode,Insert,TestTree *)
  IMPORT StdLog;

  TYPE
    Tree* = POINTER TO Node;
    Node* = EXTENSIBLE RECORD
      key-      : INTEGER;
      left, right : Tree;
    END;

    CenterTree* = POINTER TO CenterNode;
    CenterNode* = RECORD(Node)
      width  : INTEGER;
      subnode : Tree;
    END;

  PROCEDURE (t: Tree) Insert* (node: Tree), NEW, EXTENSIBLE;
    VAR p, father: Tree;
```

²'*' or '-' are export marks. Identifiers marked like this can be used from outside the module. '*' allows read/write access and '-' restricts access to read-only access. Identifiers without marks are for private use in a module only. They can not be seen from the outside.


```

BEGIN p := t;
  REPEAT father := p;
    IF node.key = p.key THEN RETURN END;
    IF node.key < p.key THEN
      p := p.left
    ELSE
      p := p.right
    END
  UNTIL p = NIL;
  IF node.key < father.key THEN
    father.left := node
  ELSE
    father.right := node
  END;
  node.left := NIL;
  node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert* (node: Tree); (* redefinition *)
BEGIN
  StdLog.Int (node(CenterTree).width);
  t.Insert^ (node) (* calls the Insert method of Tree *)
END Insert;

PROCEDURE TestTree*; (* Command which can be called as 'Trees.TestTree' *)
VAR
  tree : CenterTree;
  node : CenterTree;
BEGIN
  NEW(tree);
  tree^.key      := 1;
  tree^.left     := NIL;
  tree^.right    := NIL;
  tree^.width    := 55;
  tree^.subnode  := NIL;

  NEW(node);
  node^.key      := 2;
  node^.left     := NIL;
  node^.right    := NIL;
  node^.width    := 60;
  node^.subnode  := NIL;

  tree.Insert(node);
END TestTree;

BEGIN
  ...
  StdLog.String("Trees loaded"); StdLog.Ln
CLOSE

```

```
...
  StdLog.String("Trees removed"); StdLog.Ln
END Trees.
```

Record types and classes are final by default. Record types and classes which should be extended in the future, have to be marked by extra keywords like `EXTENSIBLE` or `ABSTRACT` as in `Node* = EXTENSIBLE RECORD ... END;`.

Some new features like the limited use of a type add new valuable hiding concepts. Annotating a `RECORD` type with the keyword `LIMITED` allows variables of this type only to be allocated in the module containing the type definition. Extensions to this type are also only allowed within the boundaries of its declaring module.

Nevertheless, Component Pascal has no means to define extra communication paths between components as e.g. in terms of extra entities like ports which allow simple connections to be established between them.

Component Hierarchies: As components are not types, it is not possible to declare variables of a component type nor to instantiate components. Consequently, component instances can not be nested into others as in other component oriented languages.

Components can refer to others by `IMPORT`-declarations (see below), but this is a uses-relationship instead of nesting components into others.

Subtyping and Dependencies: As modules are no types, subtyping between components is not declared.

If modules depend on other modules, the required ones have to be imported explicitly by an `IMPORT`-declaration. Entities, exported by a module `A` which is itself imported by a module `B`, are referred to in `B` by qualifying their names with the module name `A`, e.g. `A.x`.

Miscellaneous: Components written in Component Pascal can be used as black boxes, which can be developed independently of their use. Components can be deployed separately and used by third parties. Components required by deployed components must also be available.

When a component is loaded into memory, all components/modules it depends on are loaded first. Upon loading, code specified in a special `BEGIN`-section of the module is executed. This code is usually used to initialize the variables of the module. Circular dependencies between components are not allowed. Immediately before removing a module from memory, code specified in a special `CLOSE`-section of the module is executed.

Component Pascal is available as a .NET language. From a conceptual point of view, components in Component Pascal can be compared to assemblies in .NET.

6.1.4 Others

ACOEL: ACOEL [Sre01] is a component-oriented language similar to ComponentJ and ArchJava. Components communicate via their provided and required ports only, in ACOEL called in and out ports. In addition to interface types, ACOEL also supports function types as types for ports. Provided ports typed by interfaces are implemented by classes internal to the component instead of method blocks as in ComponentJ. These relationships are made explicit by attach statements assigning classes to provided ports. A similar attachment is needed to assign a method implemented by a component to one of its provided ports typed by a corresponding function type. Components internally declaring variables of other component types as parts are not allowed to call methods of the provided ports of their parts directly. Instead, a composite component has to declare required ports for all provided ports of the parts it wants to use. The corresponding ports of the composite component and the internal parts have to be connected. An instance of a composite component can then invoke methods of its parts by calling the corresponding methods on its own required ports. A component can extend another component. In this case, the extended component inherits all ports of its base component. It is not clear how subtyping in ACOEL really works.

Prototype Based Component Evolution: Zenger uses a prototype based approach to develop and compose components [Zen02]. As in other approaches, a component provides a set of services and can require a set of services from other components. Services are identified by the names of their types instead of port names.

In contrast to the approaches already discussed here, a component can only be represented as an extension of an already existing component. To be able to start from scratch, a special empty component exists which has to be refined stepwise. In one step, only one service can be added, except, when mixins are used (see below). Services already implemented can be overridden later on. Similar to ComponentJ, all requirements of a component have to be resolved at instantiation time. The implementation of a service can be forwarded to a service of another component.

Forwarding and mixin techniques are used to declare component hierarchies. If a component C is extended by a component D via a mixin-operation, the new component has all provided services that are provided by either C or D . It requires all services that are either required by C or D and that are not provided by C or D . If D and C both provide the same service, the implementation of C is overridden by that of D .

Component types and subtyping between components are defined similar to [SC00a].

I suppose that this kind of component oriented programming is not suited for commercial software development. Too many steps have to be done to develop a component providing several services. Which services a component provides can hardly be derived from the program code. Additionally the kind of programming is rather complicated and too far away from current programming languages as demonstrated by the following code snippets from [Zen02] which still shows simple components. Here **component** is the empty component, CustomerID, CustomerDB, StockDB and OrderDB are

types of services, `MyCustomerIDs` is an implementation for `CustomerID` and `MyCustomerDB` for `CustomerDB`. The declarations of the service types and implementations are omitted.

```
c0 = component provides CustomerID as This with new MyCustomerIDs();
c1 = c0 provides CustomerDB as This with new MyCustomerDB(This);
d0 = ...; e0 = ...;
f0 = d0 provides StockDB as This with
    (new (e0 provides OrderDB as Me with This::OrderDB))::StockDB;
```

Cells: Rinat and Smith introduce cells [RS02] as a special kind of components. Cells are containers for objects and code. In addition to attribute-declarations and operation-implementations known from classes, the implementation of cells can also contain class-implementations, references to other cells, link-commands to connect to other cells and calls to other cells's services. Cells can be registered and retrieved similar to remote objects.

A cell can communicate to other cells through so-called services and connectors. Services are single operations which can be called on a cell in a client server fashion without an extra linking process. Connectors are entities which are used for communication and import/export of classes from/to other cells. Connections via connectors have to be established explicitly using a link command. After linking, communication or import/export via the linked connectors is possible. When linking a connector of one cell to a connector of another cell, both cells can do security checks. Depending on the result, connections can be accepted or refused. In contrast to the other languages already presented, established connections can be disconnected explicitly.

Connectors consist of plugins (required operations or classes) and/or plugouts (provided operations or classes). Connectors declaring only operations as plugins and plugouts can be compared to ports in ArchJava. Connectors for classes are used to import classes from other cells which represent e.g. class libraries. Imported classes can be used and specialized in the importing cell. Class-based connectors can only be used locally. They can not be used for cells residing in different CVMs (Cell Virtual Machine).

Cells can not be nested. Thus component hierarchies can not be realized.

Cells are typed by their set of operations provided as services and their set of connectors. A subtype has at least to provide the same services and connectors as its supertype. Corresponding connectors may have more plugouts and less plugins which bears the same problems as already mentioned for ComponentJ.

6.2 Composition Languages

In contrast to component oriented languages, composition languages [WCD⁺01, Bir01, BH00, Nie99, AN01] are not designed to develop new components from scratch. Instead their focus is on composing applications and, in some approaches, new components from already existing ones delivered by third parties. Thus composition languages are

simpler than component oriented programming languages or other OO-programming languages. Their instruction sets are intensively reduced. They generally have first-class syntax and semantics to support composition operations and are typically interpreted. Some of them provide a means to hierarchically compose component instances to yield new components, even if these instances belong to flat component models.

6.2.1 Bean Markup Language

The Bean Markup Language [WCD⁺01] is a composition language designed to build applications or new JavaBeans by composing already existing JavaBeans. It is a small, simple, XML-based language allowing to instantiate beans, to manipulate their properties and fields, to call their methods, to connect beans via events, to group beans by containers and to build new JavaBeans.

The limited set of available operations is shown in the following table.

Element	Description
<bean>	Create a new bean or look one up
<args>	Specify constructor arguments
<string>	Create a new string bean or look one up
<property>	Set or get a bean property
<field>	Set or get a bean field
<event-binding>	Bind an event from one bean to another
<call-method>	Call a bean method
<cast>	Type convert a bean to be of another type
<add>	Insert a bean into a container
<script>	Defines a BML script to be used somewhere
<beanDef>	Defines a new bean
<constructorDef>	Defines a constructor for a new bean
<methodDef>	Defines a method for a new bean
<propertyDef>	Defines a property for a new bean
<eventDef>	Defines a set of related events fired by a new bean

Table 6.1: BML-Operations

The composition mechanisms provided by BML are the connection of beans via events using event adapters if needed, the nesting of beans into containers and the hierarchical composition of beans yielding new JavaBeans.

Using the <event-binding> tag, event connections are established by registering a suitable listener at an event source or by binding a script to an event fired by a bean. The script can contain e.g. arbitrary method calls to other beans. When scripts are bound to events, BML uses special event adapters which can be registered at the event source as a suitable listener and which are able to execute the corresponding script.

Beans can be added to instances of arbitrary container types like *Vector*, *JFrame*

etc. by executing the `<add>` instruction. In BML, the structure of this command is equal for all kinds of containers. When executing such an instruction, the BML interpreter/-player maps this add-command to one which fits to the type of the container the bean is added to. E.g. if the container is of type `Vector`, `addElement` is used; if it is of type `JFrame`, `<add>` is mapped to a sequence of method-calls: `getContentPane` followed by the `add`-method of the content pane. These mappings are stored in a so-called *adder registry*.

Hierarchical compositions of beans, in BML referred to as *recursive composition*, are introduced by a `<beanDef>` element. The methods, properties and events of the new bean can be defined by delegation or direct implementation. Implementation by delegation maps a property, method or event to a property, method or event of a bean the new one is composed of. Direct implementation means implementation by a BML `<script>` element which must not contain any `<beanDef>` elements. The implementation of methods by delegation may become ambiguous, if a bean provides overloaded methods and the method delegated to is only referred to by its name. Events belonging to the same event set which are fired by different internal beans can not be exported simultaneously.

BML allows some kind of macro expansion by allowing a `<bean>` tag to specify the name of a BML-file instead of a class name to be used for instantiation. This allows e.g. already pre-configured beans to be reused in other BML scripts.

In contrast to our approach which supports different component models, BML is designed to compose JavaBeans only. BML has the same shortcomings as the JavaBeans component model discussed in Section 3.1. When building composites, BML only allows one to export properties, events and methods of its parts, not whole interfaces or plugs. Similarly connections based on interfaces and plugs are not supported, only connections based on events. By referring to a bean class in the `<bean>` tag, beans being part of a composite component are already bound to a special implementation. In our approach, the type of a part can be specified only by its component interface type and thus allows implementations from different vendors to be used without problems, as far as these implementations conform to the component interface type.

The nesting of component instances into containers not explicitly supported in our approach can be modelled as follows. The container declares an optional required interface of a type identifying the component instances which can be inserted. The connect-method corresponding to this required interface refers to the `adder`-method or a sequence of methods used to insert component instances into the container.

6.2.2 Bean Plans

Birngruber and Hof introduce *bean plans* [BH00] to simplify the assembly process especially for non experts. A bean plan describes a set of pre-configured and pre-connected beans with still some degrees of freedom. Bean plans are created by developers of class libraries, so-called *bean suites*, and reflect the developers' knowledge of the underlying

bean suite and its typical composition patterns. The bean plans are provided to application programmers to simplify their assembly task. The plans relieve the application programmer from routine configuration tasks and a deep knowledge of the underlying bean suite. A bean plan can be regarded as a composition template with predefined placeholders for certain property settings and beans to be chosen by the application programmer from a predefined set of beans.

Bean plans are typically interpreted by a plan builder which guides the application programmer through the assembly process similar to a wizard. The plan builder instantiates the declared beans, configures them by setting their properties and wires them together via events according to the plan. Additionally, it adds beans to containers whenever requested by the plan. If the plan builder detects a placeholder, called an *interactive decision spot*, the tool asks the application programmer for the information needed. Decision spots can comprise one or more choices to select a JavaBean. Choices are denoted in angle brackets and are separated by “|” as e.g. in `<FlowLayout | BorderLayout | CardLayout >`. This choice means that the application programmer can select one of these three layout managers or one of their subclasses. If subclasses are not allowed, the bean-classes in the choices have to be prefixed with the keyword *fixed*. The plan builder offers all possible classes matching the choices declared by the decision spot in a user dialogue from which the programmer can select the one he wants to use. If a property value can be chosen by an application programmer, the corresponding customizer or property sheet will be shown by the plan builder to ask the programmer for a value. Once, all decision spots of a bean plan are resolved, the plan-builder produces an XML-representation of the configured and connected beans. This representation can be interpreted by an assembly tool which in turn produces the set of Java class files making up the assembled application. The creation of new beans from assembled ones is not supported.

The following example shows a bean plan from [BH00] which adds a button and scroll-pane containing a `JTable` into a `JPanel`. The background color for the `JPanel` can be chosen by the application programmer as well as a subtype of `TableModel` as the model to be used to access the data shown in the `JTable`.

```
import javax.swing.*;
import java.awt.event.* ;
import java.awt.Color;
import java.awt.BorderLayout;
import javax.swing.table.TableModel;
plan JTableView {
  Bean refreshButton = new JButton {
    text = Refresh; // set property
  }
  // ask the programmer to pick a TableModel and show
  // the appropriate customizer or property sheet of the model
  Bean tableModel = customized new <TableModel> { }
  Bean tableView = new JTable {
    autoCreateColumnsFromModel = true;
    model = tableModel;
  }
}
```

```

}
Bean jScrollPane1 = new JScrollPane {
    viewportView = tableView;
}
Bean container = new JPanel {
    layout = new BorderLayout;
    // ask the programmer for a background color
    // using the appropriate property editor
    background = <>;
    add (jScrollPane1, BorderLayout.CENTER);
    add (refreshButton, BorderLayout.SOUTH);
}
// interconnect beans using events
on refreshButton.Action.actionPerformed call tableView.repaint();
}

```

In [Bir01] the bean plans are generalized to component plans written in CoPL, a language to describe such plans. CoPL is designed to be independent from a certain component model. The components assumed are binary ones with one or more interfaces giving access to the functionality of the component. The components are self-contained that is, they do not have required interfaces. Connections between components are based on events although the event model is not actually described. A plan with no open decision spots is translated into an assembly description in CoML, an XML-based language, which is similar to BML but which does not allow to compose components to new ones. Interpreters for Java and .NET were already implemented. Cross platform composition was not yet supported.

6.2.3 Piccola

Piccola [Nie99, AN01] is a language which models components and compositions in terms of communicating *agents* which exchange *forms* over *channels*. Agents perform computations and forms are the communicated values. As is summarized in [Nie99]:

“A component is viewed as a set of interconnected agents. The interface of a component is represented as a form, a special notion of extensible records. Piccola models composition in terms of agents that exchange forms along private channels whereas higher-level compositional abstractions are introduced as sets of operators over sorts of components.”

A form is a mapping from labels to values whereas forms themselves are values as shown in the following example from [AN01].

```

baseForm =
    Text = "foo"
    Name = Text
    Size = (x = 10, y = 20)

```


In Piccola nearly everything is represented as a form:

- component interfaces (as sets of services),
- user defined services,
- arguments for services, and
- contexts.

Agents live in *contexts* which contain the services and forms available to the agent. *Services* can be invoked by agents and correspond to function calls. A hidden label belonging to the form representing a service gives access to the agent evaluating the body of the service. Services can be defined by Piccola scripts or by external components integrated into Piccola using special libraries. The following service `hello` is defined as Piccola script (see [AN01]).

```
hello() =  
    println("hello world")
```

Piccola does not expect a special component model with a predefined plugging concept. Thus, Piccola does not provide a fixed composition model as e.g. interface based connections or event based connections. Instead, Piccola allows a user to determine his own *architectural style* defining how components should look like and how such components should be plugged together. For example, components could be files, streams and filters which should be composed in a pipe and filter style. In another context, components could be AWT components composed by adding them to containers and by wiring them via events.

In a pipe and filter architecture, streams are e.g. expected to provide the services `next`, `isEOF`, and `close` and can be composed with a filter using the `'|'`-operator yielding a transformed/filtered stream. The composition operators allowed on streams must be defined by suitable Piccola scripts as shown in [Nie99]. Then these operators can be used to represent a sequence of streams and filters as e.g.

```
filteredStream = inputStream | filter1 | filter2
```

Thus these operators represent the “higher-level compositional abstractions over sorts of components” already cited above.

In the context of AWT components, components are expected to be sources of special kinds of events. In addition, such components are expected to be capable of registering suitable listeners which respond to the events fired by corresponding actions. This kind of composition is shown in [AN01]. Registering a listener at an event source is modelled by the `'?'`-operator defined for AWT components. Creation of a fitting listener is done by evaluating an expression of the form `E(R)` where `E` represents an event type, and `R` the expected response of the listener.

```
waveButton ? Action(do: duke.wave(val = 1))
```

In this example `waveButton` represents an AWT button, `Action` an event type (E) and `do: duke.wave(val = 1)` the response (R) of a suitable listener. The listener is created by evaluating `E(R)` and is registered at `waveButton` using the '?'-operator. When the `do`-script is executed, a waving duke is shown. For more details please refer to [AN01].

Although pure Piccola is not suited to simply stick components together, libraries of Piccola scripts can be created by experts, reflecting a special architectural style. Then these libraries may be used by non-experts to simply compose components conforming to one of the supported styles. An interface to Java is provided by Piccola's core libraries. The Piccola language itself is translated to an abstract machine with agents, channels and forms which implements a variant of the π - calculus [Mil91].

6.3 Architecture Description Languages

Software architecture also extensively relies on components. Architecture description languages (ADLs) like e.g. Darwin [MK96], ACME [GMW00], and WRIGHT [All97] are used to describe software systems on a high level of abstraction modelling a system by components and their interconnections using configuration descriptions. Configuration descriptions specify the components belonging to a system as well as their interconnections. Interconnections are often modelled using explicit connectors specifying communication protocols between components.

Medvidovic and Taylor [MT00] introduce a classification framework for architecture description languages to distinguish ADLs from other approaches like formal specifications, module interconnection, simulation, and programming languages. The framework also serves to classify and compare existing ADLs. The interested reader is encouraged to use this paper for further references to existing ADLs not mentioned here.

6.3.1 Darwin

Darwin [MDK94, MDEK95, MK96] is mainly designed to describe the static and dynamic architecture of a concurrent system in terms of hierarchically composed components which interact via services. Darwin supports distribution by allowing one to specify a mapping from components to machines they run on explicitly.

The interface of a component is determined by a set of provided and required services. In all approaches discussed so far, services represent units of functionality provided or required by a component. In contrast to these approaches, services in Darwin model certain kinds of communication e.g. port based communication, stream based communication etc. Thus service types are specified as tuples consisting of a communication type and the type of the data/messages to be transmitted. The service types are

implemented by C++ templates which are expected to be available at each platform a component runs on. The following example shows a part of the C++ template for the port type from [MDK94]. Port objects are queues of messages of a type T.

```
template <class T>
  class port: public portbase {
  public:
    void in (T &msg);      // receive message of type  T into msg
    void out (T &msg);     // synchronous send msg of type  T
    void send (T &msg);    // asynchronous send msg of type  T
    ...
  };
```

Components can be either primitive or composite. Primitive components are implemented by C++ classes and are specified in Darwin as component declarations containing only the declaration of the provided and required services (see component `filter` below). Composite components consist of a set of variables representing interconnected component instances which are wired/bound together via their provided and required services. Services of internal constituents can be bound to services of the composite component thus making them available to the outside. Binding can only be done, if the communication type (port, stream etc.) and the message type (T) are the same for both services to be bound. Services of composite components may only be specified by a name, a type can be omitted. In this case, the type is inferred from the type of the internal constituent bound to the service of the composite component.

The following example from [MDEK95] shows a pipeline of filters of variable length n , where the output service of one filter is bound to the input service of the next filter. The communication pattern used is stream based communication and the messages passed are of type `char`. Thus the service type for the input and output service of a filter component is `<stream char>`.

```
component filter {
  provide output <stream char>
  require input  <stream char>
}

component pipeline (int n) {
  provide output;
  require input;

  array F[n] : filter;
  forall k : 0..n-1 {
    inst F[k] @ k + 1;
    when k < n-1
      bind F[k + 1].input -- F[k].output;
  }
  bind
    F[0].input -- input;
    output -- F[n - 1].output;
}
```

Components can be parameterized, as e.g. `pipeline` by `n` in the example above, to allow families of equal architectures to be defined. Distribution to different machines is done by assigning an integer number to a variable representing a component instance as in `inst F[k] @ k+1;`. The integer identifiers are mapped to real machines by the Regis runtime system, executing Darwin configurations.

Darwin supports the declaration of components for lazy instantiation. That is, a component is not instantiated until one of its services is called the first time.

In contrast to our approach, Darwin can not be used to create new components from existing ones which can then be deployed and reused independently. Instead, Darwin is used to master the complexity of an application by describing it as a hierarchical composition of interconnected component instances. Thus Darwin does not address substitutability of components nor subtyping between components.

Component services represent communication patterns instead of functionality. Optional requirements can not be declared. One required service may only be bound to one provided service. Higher level entities grouping several services can not be declared, as e.g. a port communication and a corresponding event communication which could be required from the same communication partner. In our approach, this feature is addressed by plugs. Primitive components are restricted to C++ classes. Components of industrial component models can not be integrated.

6.3.2 Acme

Acme [GMW00] is an architecture description language developed to provide a minimum common basis for many diverse ADLs. To support the various peculiarities of the different ADLs, Acme allows ADL specific information to be specified additionally. Such information is left uninterpreted in Acme, but can be interpreted by ADL specific tools. Such ADL specific tools have to use Acme as their common basis that is, they must be able to read and write Acme descriptions.

The core elements of Acme are *components*, *connectors*, *ports*, *roles*, *systems*, *representations*, and *representation maps* (rep-maps for short).

Components model computational elements which can communicate to their environment through a set of ports. Ports identify the points of interaction of a component to its environment and can e.g. represent single operations, a set of related operations or an event multicast interface.

Connectors are entities in their own right. They model communication between two or more components. Every connector declares a set of roles which identify the participants involved in a communication via this kind of connector. Connectors may be RPC connectors with roles caller and callee, pipe connectors with roles reading and writing, message passing connectors with roles sender and receiver etc. Components can be interconnected via a connector by attaching component ports to roles of the connector.

Systems are built from sets of components and connectors related by assigning component ports to connector roles. The following example from [GMW00] shows a simple

Acme system consisting of a client component, a server component and an RPC connector which connects client and server:

```
System simpleCS = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles {caller, callee} }
  Attachments : {
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee }
}
```

To enable hierarchical structures in Acme, components and connectors can be represented by one or more lower level descriptions called *representations*. *Rep-maps* are needed to define, how these internal representations are mapped to the interface of the entity they refine. That is e.g., which ports of which internal components represent external ports. The following shortened example from [GMW00] shows the client-server system from above with a refinement `serverDetails` for the server component. The server component consists of three components, a `connectionManager`, a `securityManager` and a `database`. The port `externalSocket` of the `connectionManager` is mapped/bound to the server's port `receiveRequest`.

```
System simpleCS = {
  Component client = { ... }
  Component server = {
    Port receiveRequest;
    Representation serverDetails = {
      System serverDetailsSys = {
        Component connectionManager = {
          Ports { externalSocket; securityCheckIntf; dbQueryIntf } }
        Component securityManager = { ... }
        Component database = { ... }
        Connector SQLQuery = { ... }
        ...
        Attachments {
          ...
        }
      }
      Bindings { connectionManager.externalSocket to server.receiveRequest }
    }
  }
  Connector rpc = { ... }
  Attachments { client.sendRequest to rpc.caller ;
                server.receiveRequest to rpc.callee }
}
```

These core elements of Acme can be enriched with additional information through *properties* which are uninterpreted in Acme. Every property has a name, an optional type and a value. If a client expects a special request rate at its port it would e.g. attach the following property specification to its port:

```
Properties requestRate : float = 17.0;
```

These additional properties can be ADL specific and interpreted by tools for other ADLs.

Constraints can also be attached to the core elements. Constraints can be either *invariant* or *heuristic*. Invariant constraints must be preserved whereas heuristic constraints should be preserved, but may be violated. For further details on constraints please refer to [GMW00].

To enable the reuse of architectural elements, Acme allows a user to define types of components, connectors, ports and roles called *structured types*. Each type definition specifies a type name, its substructure, properties and constraints. An instance of this type must at least contain all substructure elements and properties and has to respect the constraints. In addition to structured types, styles can be declared, called *families* in Acme, which specify a whole set of similar systems. The following example shows both approaches, a family of Pipe and filters using certain component and connector types. The system simplePF is defined as an instance of the family PipeFilterFam.

```
Family PipeFilterFam = {
  Component Type FilterT = {
    Ports { stdin; stdout; };
    Property throughput : int;
  };
  Component Type UnixFilterT extends FilterT with {
    Port stderr;
    Property implementationFile : String;
  };
  Connector Type PipeT = {
    Roles { source; sink; };
    Property bufferSize : int;
  };
  Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
  Invariant Forall c in self.Connectors @ HasType(c, PipeT);
}

System simplePF : PipeFilterFam = {
  Component smooth : FilterT = new FilterT
  Component detectErrors : FilterT;
  Component showTracks : UnixFilterT = new UnixFilterT extended with {
    Property implementationFile : String = "IMPL_HOME/showTracks.c";
  };
  // Declare the system's connectors
  Connector firstPipe : PipeT;
  Connector secondPipe : PipeT;
  // Define the system's topology
  Attachments { smooth.stdout to firstPipe.source;
    detectErrors.stdin to firstPipe.sink;
    detectErrors.stdout to secondPipe.source;
    showTracks.stdin to secondPipe.sink; }
}
```

6.3.3 Wright

Wright [All97] is an architecture description language which does not only describe the static architecture of a system, but also includes semantic descriptions for its basic language elements.

As Acme, Wright knows components, ports, connectors, and roles. Systems in Acme correspond to *configurations* in Wright. Hierarchies are supported by allowing the behavior of components and connectors to be further refined by architectural sub-descriptions.

Components in Wright are defined by their ports, building the interface of the component to its environment, and by a *computation* which defines the behavior of the component. The computation can be either defined by a subsystem (see above) or directly by a CSP³-like specification [Hoa85]. This kind of specification allows a user to describe behavior in terms of processes engaged in events. The behavior of ports is also defined in a CSP-like notation. If a computation is specified by a subsystem, a binding has to map ports of subcomponents to ports of the refined component like in Acme, e.g. `C.Combined = Service` (see [All97], p. 47). `Combined` is a port of the subcomponent `C` which is mapped to the port `Service` of the enclosing component.

Connectors are defined by roles and *glue*. While roles have the same meaning as in Acme, glue is unknown to Acme. Glue specifies, how the different roles, identifying participants in the communication via this connector, are coordinated to achieve the overall cooperation. The behavior of roles and glue is specified in the same CSP-like notation as computations and ports.

The following slightly modified example from [All97] shows a Wright configuration with three filter components and one pipe connector. The configuration contains one instance of every filter component as well as three instances of the pipe connector. The first filter (`Split`) is connected via pipe `P1` to the filter `Upper` converting lower case to upper case. Via `P2`, `Split` is also connected to the filter `Merge`. `Upper` and `Merge` are connected via `P3`. `Split` splits the input stream into two streams. One is sent to `Upper` which capitalizes every character and in turn sends the resulting stream to `Merge`. The second stream is sent from `Split` to `Merge` without being modified. `Merge` recombines both streams to one output stream. As in Acme, the connections between components and connectors are described by attachments which relate ports of components to roles of connectors.

```
Configuration Capitalize
  Component UpperCase
    Port Input  ...
    Port Output ...
    Computation ...

  Component SplitFilter
    Port Input = DataInput
```

³CSP is an abbreviation for “Communicating Sequential Processes”.

```

Port Left  = DataOutput
Port Right = DataOutput
Computation ...

Component MergeFilter
  Port Left  ...
  Port Right ...
  Port Output ...
  Computation ...

Connector Pipe
  Role Source ...
  Role Sink   ...
  Glue ...

Instances
  Split : SplitFilter
  Upper : UpperCase
  Merge : MergeFilter
  P1, P2, P3 : Pipe

Attachments
  Split.Left as P1.Source
  Upper.Input as P1.Sink
  Split.Right as P2.Source
  Merge.Right as P2.Sink
  Upper.Output as P3.Source
  Merge.Left as P3.Sink
End Capitalize.

```

In the example above all behavior specifications of ports, roles, computations and glue are only indicated by '...' except for the ports of the component `SplitFilter`. These ports are defined by the interface types `DataInput` and `DataOutput`, special types declaring reusable input and output behavior for filter components. In general, Wright allows one to define arbitrary *interface types* to specify certain kinds of behavior. Behavior specified by interface types can then be used for various ports and roles by assigning this type to a certain port or role as shown above for the ports `Input`, `Left` and `Right`. Thus, instead of specifying each port (or role) of a component (connector) having the same behavior by a separate CSP-expression, the same interface types can be assigned specifying this behavior only once. A typical interface type for input ports of filter components as well as a typical interface type for output ports of filter components is shown below.

Interface Type $DataInput = (\overline{read} \rightarrow (data?x \rightarrow DataInput$
 $\quad \square \text{end-of-data} \rightarrow \overline{close} \rightarrow \S))$
 $\quad \square (\overline{close} \rightarrow \S)$

Interface Type $DataOutput = (\overline{write!x} \rightarrow DataOutput) \square (\overline{close} \rightarrow \S)$

knows and respects all definitions and constraints of the superstyle and may add additional constraints.

Wright defines several consistency checks for the various elements of a configuration. Based on the CSP-specifications of ports and computation, Wright provides a test which checks, whether a computation is consistent with its ports that is, whether a computation models all interactions correctly. Another kind of check validates, whether a port may be attached to a role. An attachment is possible, if 1) the port handles all observed events specified by the role and 2) if the port chooses to initiate an event, the port must select one that is specified by the role. The port may observe other additional events or may disallow options for initiating events which are permitted by the role. Other tests concern e.g. whether critical attachments are missing or whether connectors are deadlock-free.

6.4 The Unified Modeling Language 2.0

The Unified Modeling Language (UML) is a visual modeling language for describing software systems. It can be used to specify a system, to document it and to visualize its artifacts as well as their dependencies. Using UML, the static structure as well as the dynamic behavior of a system can be described.

Version 2.0 of UML comes with several new diagrams supporting especially component based development. Unfortunately, the official document which can be downloaded at the moment “Unified Modeling Language: Superstructure version 2.0, formal/05-07-04, August 2005” still contains some inconsistencies and obscurities. In the following we shall present the main concepts described so far and will point at obscurities which yet have to be eliminated.

Then we shall compare our approach to that of UML 2.0 (especially to composite structures) as far as it is possible with respect to the existing obscurities.

6.4.1 Components

In UML 2.0 a component is regarded as an encapsulated, modular unit which communicates to its environment only through well-defined interfaces. A component may provide interfaces to its clients and it may be dependent on other elements of a system. These dependencies are specified in terms of required interfaces.

The set of interfaces provided and required by a component is regarded as the type of the component. A component may be replaced by another component, if the two components are type conformant. Additionally, a component may be used in different contexts as far as these contexts conform to the contract specified by the provided and required interfaces of the component. Unfortunately, the meaning of *type conformance* in the context of *component types* is not defined precisely. The specification only states the following ([UML05] p.142):

“A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant.”

This definition particularly does not clarify how to handle required interfaces of a component. Must a type conformant component use the same set of required interfaces or is a subset sufficient? What about the types of the interfaces etc? For a discussion on this topic please refer to Section 4.3.2.

In the UML standard, classes and interfaces can be represented by different classifiers and related by an implements relationship. But it is not possible to separate the declaration of the interface of a component to its environment (as represented by the set of its provided and required interfaces) from its implementation. A component is identified by a single classifier⁶. The UML standard profile L2 has to be used for applications which need to differentiate between the interface of a component and its various implementations. In this profile two stereotypes << specification >> and << realization >> are defined referring to component interfaces and component implementations.

Every provided/required interface of a component is regarded as being implemented/used by the component or as being exposed by one of its ports. Ports are an additional means to explicitly define named interaction points of a component which may expose one or more interfaces (provided and/or required). Since a component may form the abstraction for a set of classifiers that realize its behavior, a provided/required interface need not be implemented/used directly by the component; it may be implemented/used by one of its realizing classifiers⁷.

For every component artifacts that implement the component may be specified. Typical artifacts are e.g. JAR-files, EXE-files and DLL's. Artifacts should be capable of being deployed and re-deployed independently, for instance to update an existing system.

Several components may be used to build a new component or a subsystem. The used components are “wired” together via their required and provided interfaces. This wiring can be structurally defined by using dependencies between component interfaces and is done on classifier level, not on instance level (see Figure 6.3).

In the following some UML component diagrams are shown. They are taken from the current UML-specification. Some of them are adopted slightly to our needs.

The first diagram shows different levels of detail which may be used to specify a component. The second diagram shows another representation equivalent to the first one. Only the indication of the used artifacts is missing in diagram 6.2. The third diagram (Figure 6.3) shows the wiring between components using dependencies⁸ be-

⁶In UML a classifier denotes a “collection of instances that have something in common”. Classifiers include among others interfaces, classes, datatypes, and components.

⁷It does not become clear whether realizing classifiers may only be classes in the sense of OOP or whether they also comprise subcomponents. We assume the latter case especially because components may be assembled from other (sub-)components.

⁸For the semantics of dependencies please refer to [Fow04, JRH⁺04].

tween interfaces on structure diagrams (see above). This diagram also shows a mapping from the external view (exposed interfaces) of a component to its internal view (realizing classifiers). The realizing classifier `OrderHeader` exposes its required interface `AccountPayable`.

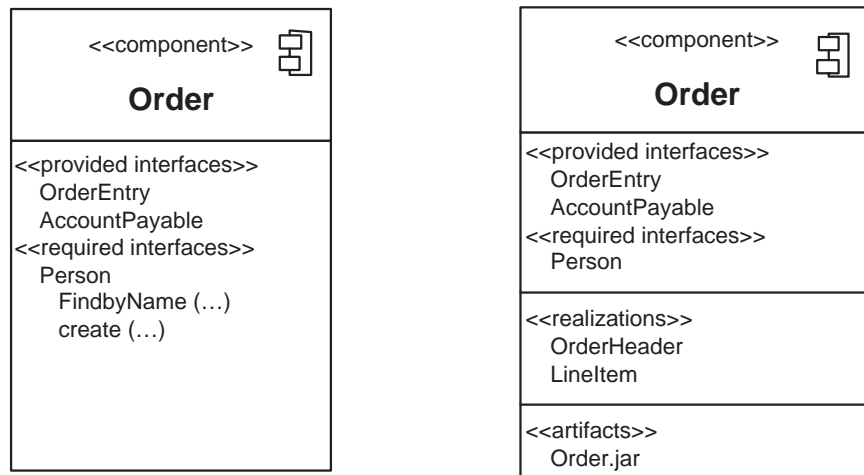


Figure 6.1: Black- and White-Box Views of a Component (taken from [UML05] with slight modifications)

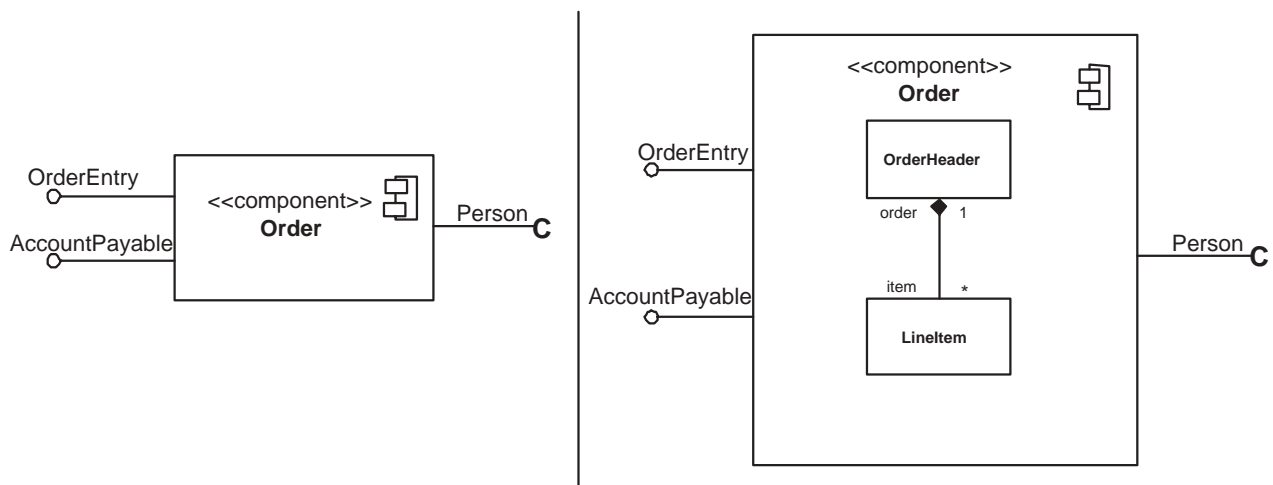


Figure 6.2: Another Representation of the Black- and White-Box View of a Component (taken from [UML05] with slight modifications)

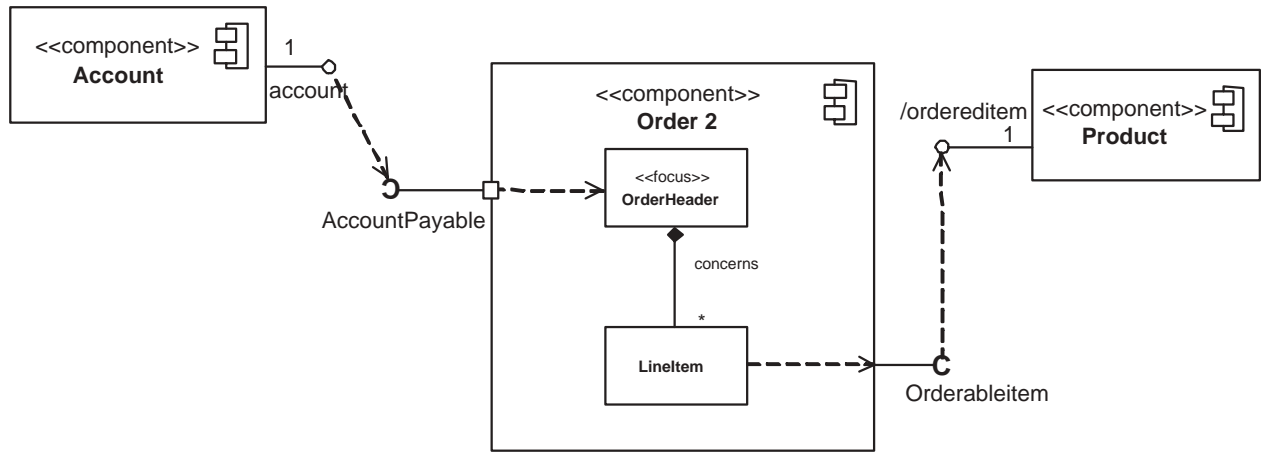


Figure 6.3: Wiring through Dependencies on a Structure Diagram (taken from [UML05])

6.4.2 Internal Structure of a Component

Structure diagrams as described in Section 6.4.1 are useful to show which components are needed in a system and how they depend on each other on the classifier level. But these diagrams are not appropriate to express all the information needed for interconnected components on the instance level. Let us have a look at the following example.

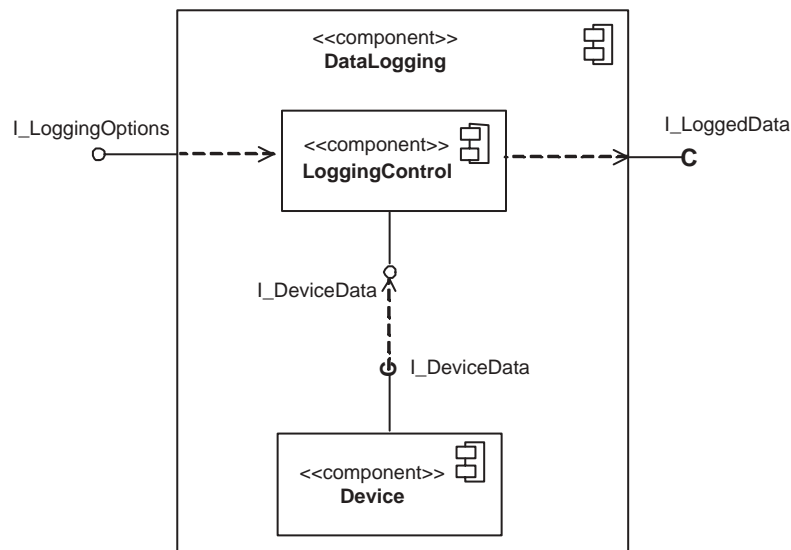


Figure 6.4: Data Logging Component

A data logging component collects data from several connected devices and stores these data to e.g. a database via its required interface `I_LoggedData`. It collects data only from one device which can be selected by a client via the provided interface `I_LoggingOptions`. This interface also provides operations to start and stop data logging.

The data logging component is realized by two other components: `LoggingControl` and `Device` which are wired together through their provided and required interfaces `I_DeviceData`.

This representation of the component does not show implementation details as for example that there are only two devices available (as can be expressed by a different representation as shown in Figure 6.5) or even that there exists one device for selecting data from a supplier belt and one for data from a robot (see Figure 6.6). In the last case both devices play different roles in the containing component.

To be able to describe such details, an internal structure consisting of **parts** and **connectors** can be defined for a component. A part constitutes a named set of instances typed by the same classifier and playing the same role in the containing component. Instances represented by a part are owned by the enclosing component instance. In Figure 6.5 there exist two parts: one with name `LC` and typing classifier `LoggingControl` and one with name `controlledDevice` and typing classifier `Device`. `LC` denotes only one instance of `LoggingControl` whereas `controlledDevice` denotes two instances of `Device` indicated by the multiplicity declaration `[2]` following the typing classifier `Device`. The multiplicity declaration for a part restricts the number of instances belonging to this part.

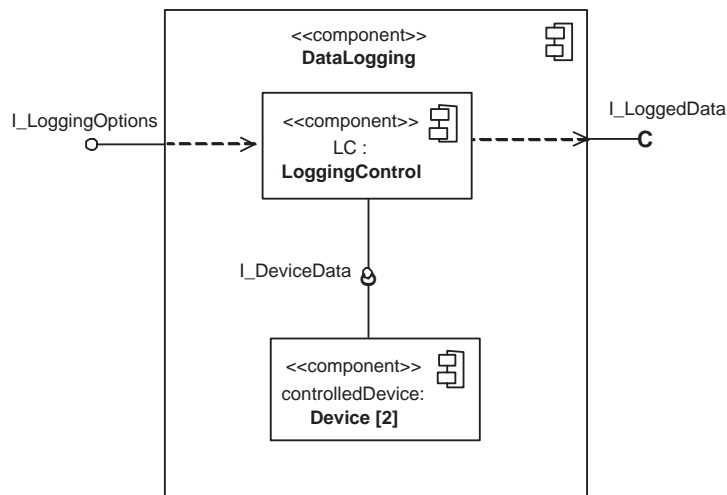


Figure 6.5: Data Logging Component using Parts

Figure 6.6 presents a more detailed view by introducing three parts: one for the instance of a `LoggingControl` component and two for the different devices.

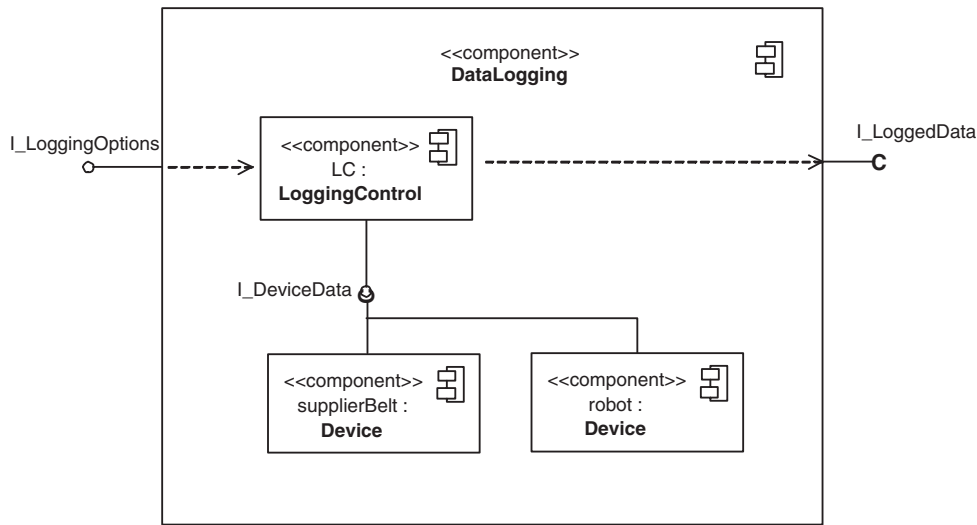


Figure 6.6: Detailed View of the Data Logging Component

Parts may be connected by **assembly connectors**. An assembly connector is a connector wiring a required interface or port(see below) to a provided interface or port. A request originating in a required interface of a connected component instance corresponding to one part results in the invocation of the corresponding operation of the connected provided interface of the component instance corresponding to the other part.

Besides parts, **ports** may be declared defining distinct interaction points to the environment of the component instance. Ports can be used to completely isolate the internals of a component from its environment. Through ports, functionality realized or needed by internal parts may be exposed to the environment.

Ports may be typed by one or more interfaces which are provided and/or required. A port which is only typed by one interface is called **provided port** or **required port** depending on whether the exposed interface is provided or required. Ports typed by several interfaces are called **complex ports**. Ports allow one to distinguish interfaces by introducing explicit port names and to logically group interfaces. Especially as multiplicities may be declared for ports there may exist several ports typed by the same interface(s), although this is not explicitly stated in the UML specification.⁹

To express that interfaces exposed by ports are realized or used by special parts, **delegation connectors** are introduced.

“p. 150: A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the components parts. It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling.

⁹ If a component uses ports to communicate with its environment, it is not clear how the different ports are considered in the type of a component, especially, if several of these ports have the same interface type. The type of a complex port is not clarified either. In the examples presented in the UML-specification only the provided interface of a port grouping one provided and one required interface is denoted as its type.

p. 151: Delegation connectors are used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. The word delegation suggests that concrete message and signal flow will occur between the connected ports, possibly over multiple levels. It should be noted that such signal flow is not always realized in all system environments or implementations (i.e., it may be design time only)."

If we modify Figure 6.6 such that the data logging component uses ports and delegation connectors to expose its provided and required interfaces and to express the delegation to / from internal parts, we obtain the following figure.

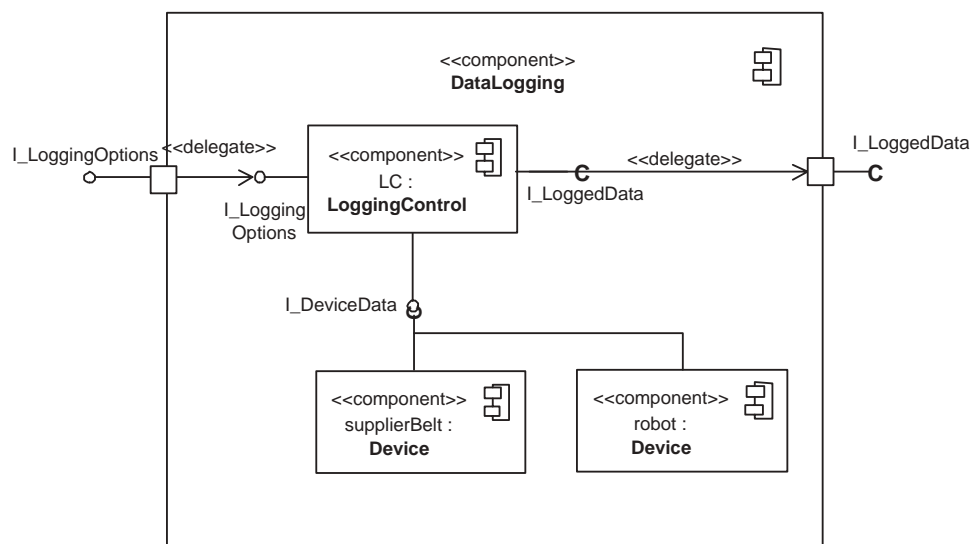


Figure 6.7: Data Logging Component using Parts, Ports and Connectors

It is left open how complex ports may be connected. Do they have to be connected to a complementary complex port that is, has the port to be connected to expose the same set of provided and required interfaces, but with reversed roles concerning provisions and requirements? Or which restrictions have to be adhered to? How can connections between complex ports be represented graphically? May interfaces belonging to a complex port be connected to interfaces of different ports? Whereas the UML-specification states the following with respect to complex ports (page 176):

"The required interfaces characterize services that the owning classifier expects from its environment and that it may access through this interaction point: Instances of this classifier expect that the features owned by its required interfaces will be offered by one or more instances in its environment. The provided interfaces characterize the behavioral features that the owning classifier offers to its environment at this interaction point. The owning classifier must offer the features owned by the provided interfaces. "

Rumbaugh, Jacobson, and Booch [RJB05] state on page 526:

“Two internal ports connected together must be of complementary types, because a request sent by one is serviced by the other. Two types are complementary if the required services of each are a subset of the provided services of the other.”

If a component containing parts and ports is instantiated, instances corresponding to all parts are also instantiated, immediately or at a later time. If the component instance is deleted, all instances corresponding to its parts are deleted with it. That is, a component containing parts has the responsibility for the existence and storage of the instances corresponding to the parts (whole/part relationship). An instance corresponding to a part may be included in at most one containing component instance at a time. Similarly upon component instantiation time instances for all ports are created. Instances of ports are referred to as **interaction points** and provide unique references.¹⁰ The interaction point object must be an instance of a classifier that realizes the provided interfaces of the port. Also links corresponding to the connectors wiring different parts or ports and parts are created upon instantiation of the containing component. A link defines a communication path between the connected entities (instances corresponding to ports / parts). A link may be realized in many ways: e.g. by a simple pointer or even by a complex network connection.

The following figure summarizes almost all of the concepts introduced to describe the internal structure of a component.

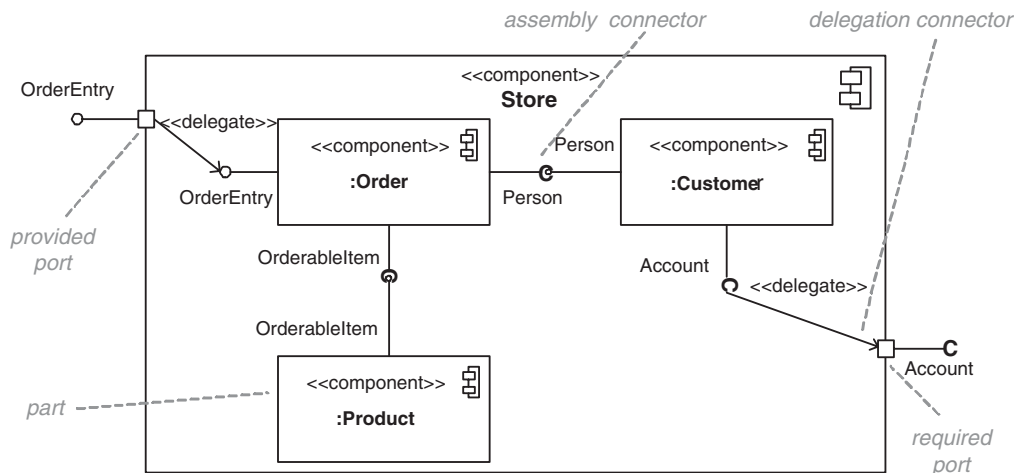


Figure 6.8: Internal Structure of a Component containing other Components as Parts (taken from [UML05])

¹⁰ The semantics of an interaction point does not become clear. There should e.g. exist a link from an interaction point to the instance of the containing classifier/component: “A link from that instance to the instance of the owning classifier is created through which communication is forwarded to the instance of the owning classifier or through which the owning classifier communicates with its environment.” But as component instances may be abstract in the sense that “an object specified by the component does not exist, that is, the component is instantiated indirectly, through the instantiation of its realizing classifiers or parts” what does a link to it mean?

6.4.3 Comparison of the UML- and our UCM-Approach

In the following, the concepts of UML concerning components are compared to our UCM approach.

Concept	UML	UCM
Specification of the interface of a component	In contrast to interfaces which are implemented or used by classes, basically there does not exist any explicit notion for the interface of a component to its environment e.g., as a classifier in its own right apart from its implementation. The interface of a component is implicitly defined by the set of its provided and required interfaces which can be represented in the different black- and white-box views. Only when using the standard profile L2 it is possible to distinguish between the specification of a component's interface and its realizations by using the stereotypes << specification >> and << realization >>.	The interface of a component is declared apart from an implementation of a component. Several component implementations can be related to the same component interface specification by an implements relationship.
Type of a component	As described on page 259 the UML specification states that the type of a component is regarded as the set of its provided and required interfaces. Unfortunately it is not clear, how far ports are incorporated in this definition especially if several ports of a component are typed by the same interface or if a component has complex ports. The meaning of type conformance is not clear either. For further discussions see page 259 and 264.	The type definition of a component supports the occurrence of several interfaces of the same type. They are distinguished by their (service-) names. In contrast to UML, our type definition (definition 4.3.8) also takes into account the grouping of services by plugs (which can in some sense be compared to complex ports; see below) as well as constraints concerning interconnections. Type conformance is precisely defined by our subtype relation for component interfaces.

Concept	UML	UCM
Realizations	<p>The set of classifiers that realize the behavior of a component. It remains unclear whether realizing classifiers may only be classes in the sense of OOP or whether they also comprise subcomponents. But in the context of components being assembled from subcomponents, we assume the latter case.</p>	<p>The set of classifiers denoting subcomponents realizing the behavior of a component corresponds in our approach to the set of component implementations to be used for the parts a composite UCM-component is built from (see Chapter 4 and Section 4.2.2). For an atomic UCM-component there exists only one realizing classifier denoting the implementing component which belongs to an industrial component model. This classifier is referred to by the entry 'ImplementingComponent' in the component implementation description needed for our model. This entry contains the necessary information for the lookup and instantiation of the needed component.</p>

Concept	UML	UCM
Artifacts	In the context of components artifacts are typically physical entities that implement the component. Typical artifacts are e.g. JAR-files, EXE-files, DLL's, class files and source files.	Something like artifacts only occurs in component implementations of atomic UCM-components. The entry 'ImplementingComponent' may refer to the name of a Java class identifying a JavaBean. Normally this entry does not refer to the physical entity containing the code of the needed (industrial) component as e.g. the name of an exe-file containing a COM-server managing a needed COM-class or a Jar-archive containing the class-files implementing a needed JavaBean or EJB component. Instead, this entry is used to provide the necessary information for the lookup and instantiation of the needed component. It therefore refers to the needed component by some kind of indirection as e.g. by using a class ID for an installed COM component or the JNDI-name for an already deployed EJB.

Concept	UML	UCM
Part	<p>A part represents a set of instances of one component that are owned by an enclosing component instance and play the same role therein (see e.g. Figure 6.5). The cardinality of this set is restricted by the multiplicity denoted for this part. A component containing parts has the responsibility for the existence and storage of the instances corresponding to the parts (whole/part relationship). An instance corresponding to a part may be included in at most one enclosing component instance at a time.</p>	<p>Our parts may be compared to parts in UML annotated with a multiplicity of one. A part therefore corresponds to exactly one instance of a component. In contrast to UML where parts are typed by classifiers which can not distinguish between the interface of a component and its implementation (see Section 6.4.1), our parts are always typed by component interface types. The component used at instantiation time is either selected according to the assignment of component implementations to component interfaces in the 'ImplementationBinding' section of the component implementation, if present, or else by the runtime system (see Section 4.2.2).</p>
Provided / required ports	<p>Provided resp. required ports are typed by only one interface. A port defines a distinct interaction point of a component where the component communicates to its environment or to internal parts. Ports separate the internals of a component from its environment. Ports allow one to distinguish interfaces of the same type by introducing explicit port names.</p>	<p>Our services may be compared to simple (i.e. non-complex) ports. Provided services correspond to provided ports and required services to required ports. For every provided service of a component at runtime there exists an interface object handling the requests. This object is created by an instance of an atomic UCM-component (see Section 4.2.2) which implements this interface and which may be nested multiple levels deep within the enclosing (composite) component.</p>

Concept	UML	UCM
Provided / required ports (continued)		<p>A reference to this interface object is made available to the environment of the containing component instance.</p> <p>There does not exist a separate interaction point object that hides the interface object from the environment and performs explicit delegation of requests to this interface object.</p>
Complex ports	<p>A complex port is a port which groups a set of provided and /or required interfaces to build a distinct interaction point. At instantiation time, an interaction point object providing a unique reference is created <i>“that must be an instance of a classifier that realizes all the provided interfaces of the port”</i>.</p> <p>Although the terms <i>“interaction point object”</i> and <i>“providing a unique reference”</i> used in the UML-specification suggests an object in the sense of OOP, it is not really clear whether this ‘object’ may also be an ‘abstract’ component instance where the interfaces are realized by internal parts.</p>	<p>Our plugs are also used to logically group a set of provided and required interfaces. But plugs allow a grouping, in which an interface may occur more than once. This is due to the fact that plugs group services (i.e. <i>named</i> interfaces allowing to distinguish interfaces of the same type) instead of merely interfaces. In UML a plug would correspond to a grouping of simple ports.</p> <p>If a client wants to access an operation provided by one of the services grouped by a plug, it gets access to the interface object corresponding to this service. There is no need for providing an object that implements <i>all</i> of the provided services grouped by the plug as seems to be intended by UML. Such a demand would cause a problem for plugs grouping several services of the same (interface) type.</p>

Concept	UML	UCM
Complex ports (continued)	Also the type of a complex port remains obscure. In the examples presented in the UML-specification only the <i>provided</i> interface of a port grouping one provided and one required interface is denoted as its type. The required interface is not taken into account. The connection between complex ports remains obscure, too. For further details please refer to page 265.	In contrast to UML, we exactly define the type of a plug which takes also required services into account. Based on this definition, we declare when two plugs may be connected to each other and how this may be achieved.
Assembly connector	An assembly connector connects a required interface or port of a part to a provided interface or port of another part.	The definition of an assembly connector is equivalent to our definition of an internal connection between a required service of one part and a provided service of another part in the section 'InternalConnections' of a component implementation.
Delegation connector	A delegation connector is used to express that an interface of a component exposed by one of its ports is realized or needed by a special part of the component.	The concept of a delegation connector corresponds to our export of services. An export definition links a service of the component interface to a service of a fitting type of one of the component's parts.

In the following we relate the graphical representation of the component CP_DataLogging introduced in example 4.2.1 and already shown in Figure 4.17 to an almost equivalent representation in UML. For the sake of simplicity, the graphical representation of this component using our notation is repeated below.

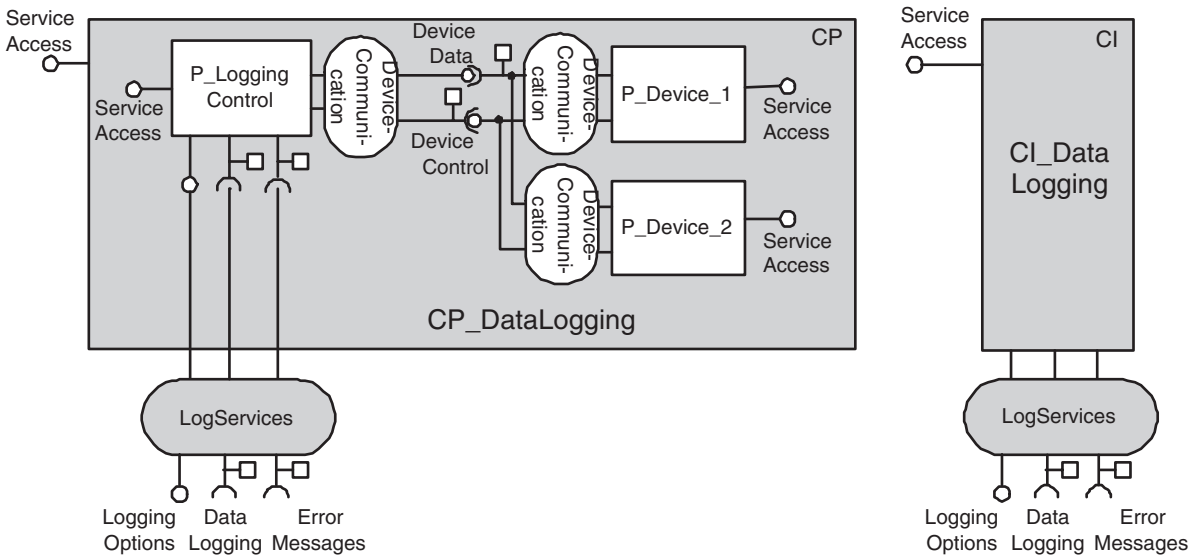


Figure 6.9: Composite UCM-Component CP_DataLogging with its Component Interface CI_DataLogging

CP_DataLogging contains the parts P_LoggingControl, P_Device_1 and P_Device_2. From example 4.2.1 we know that P_LoggingControl is of type CI_LoggingControl which is implemented by CP_LoggingControl. P_Device_1 and P_Device_2 are both of type CI_Device which is implemented by CP_Device.

According to the naming conventions used throughout example 4.2.1, the types of all services can be derived from their names by prefixing the service names by I_. Only the service ErrorMessage is of type I_Error instead of I_ErrorMessages. The service types were omitted for the sake of simplicity.

Mapping our representation to an (almost) equivalent representation in UML we obtain Figure 6.10. Since DeviceCommunication, the name of the used plugs, was too long, we changed it to DeviceCM in the corresponding UML-diagram.

Services are represented by ports. Service names are mapped to port names. The interfaces are annotated by their type names.

In our approach parts are always typed by component interfaces which is not possible in standard UML because component interfaces can not be declared apart from component implementations (see 'Specification of the interface of a component'). But as in our example for every component interface an implementation is defined which has to be used for the parts typed by these component interfaces, we can use these implementations as classifiers for the parts in UML.

Since plugs group services instead of interfaces which would refer to a grouping of simple ports instead of merely interfaces in UML, plugs can not always be mapped to equivalent complex ports. A plug grouping two or more services of the same type can not be mapped to a complex port. In our example however, where all plugs group services of different types, an unambiguous mapping to complex ports is possible.

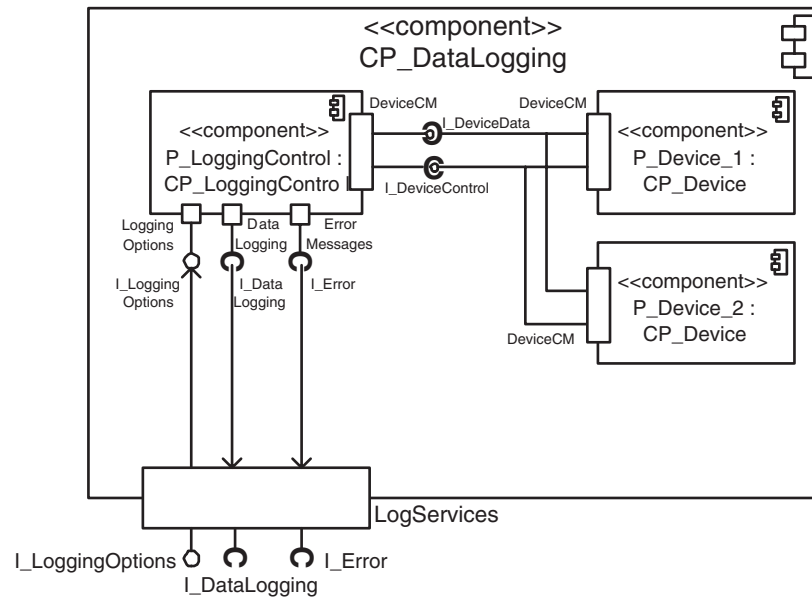


Figure 6.10: CP_DataLogging represented in UML Notation as a Composite Structure Diagram

Although in the superstructure document of UML 2.0 examples are missing which use delegation connectors from / to complex ports or assembly connectors between complex ports, we suppose we correctly represented such connectors in Figure 6.10.

Constraints on interconnections like a lower and upper limit on the number of connections to a required service as represented in Figure 4.25 or a constraint like *Different Service Providers* represented in Figure 4.21 have no counterpart in UML.

Chapter 7

Summary and Perspectives

This chapter summarizes the work at hand and outlines future work.

7.1 Summary

The importance of components for software development in our world of high productivity with a focus on maximizing profit is recognized. The vision of component based software development which promised to decrease time to market, to increase reliability and to reduce development costs by just composing prefabricated, reusable, well-tested components to new applications has already partly come true by the various component models introduced by industry in the last years. A lot of prefabricated components especially for graphical user interfaces were developed. A company using such prefabricated components, however, has to use the component model the components were constructed from. Unfortunately, the various component models differ a lot as we could see in Sections 2.2 and 3.1. A company which wants to use different component models simultaneously for its various applications has to investigate time and money for a well-trained staff.

Thus, our first goal was to develop a unifying component model which comprises the main features of current industrial component models and which allows us to integrate the existing models. We called our unifying component model *UCM* (Section 4.1). Now less skilled programmers can refer to UCM only and need not worry about the different underlying industrial component models. Prefabricated components from existing component models can be integrated and need not be re-implemented.

UCM: In addition to the main concepts of the industrial component models, UCM (see Section 4.1) provides some other useful features like plugs and a certain kind of alias control. The main characteristics of the model are summarized below.

- A component communicates with its environment through a well-defined interface only.

- The component interface is specified apart from the component implementation. Several components can implement the same component interface.
- The component interface is specified by a set of provided and required services as well as a set of plugs. Services including required ones are already known e.g. by CCM as facets and receptacles. Plugs, as groups of semantically related services, acting as a unit for interconnection and especially supporting bi-directional connections between two parties are new, to the best of our knowledge.
- In contrast to other models, we distinguish optional and mandatory required services. Whereas mandatory required services are services a component needs to work properly, optional required services are mainly used for notification purposes and possible extensions.
- Our required services have an explicit means to declare a lower and upper limit on the number of connections established to them.
- Connections to a required service are established via an explicit connection point object which belongs to the required service. This object implements the methods available to establish or release a connection. Every required service may declare its own methods for connection. This is essential to be able to integrate existing concepts as e.g. event connections in the JavaBeans component model, which defines its own standard methods to register listeners. As we want to support the various approaches of the different component models, we can not dictate how connect methods should look like.
- Our model allows us to define a pair of connect- and disconnect-methods together with their default parameter-values which can be used by tools to establish and to release connections without user interaction.
- Our model supports a certain kind of alias-constraint which ensures that each required service of a component instance belonging to a specified set is connected to a different service provider.
- Components belonging to industrial component models can be integrated as atomic UCM-components.

This component model simplifies composition by an easy means to connect component instances via services and even plugs. But there is still a gap between this approach and the goals of CBD, namely to simply stick components together to build a new application. Still programming languages have to be used to connect component instances.

Therefore we introduced a simple composition language which allows us to easily declare interconnections by abstracting from the programming language details. The language is enhanced by an easy means to compose components to new ones hierarchically. This helps us to reduce the complexity of building an application and thus further contributes to the goal of simplifying the process of building a new application.

A Composition Language supporting Hierarchical Composition: Our composition language which allows us to connect component instances via services and plugs and to compose UCM-components hierarchically to new ones, has the following characteristics (see Figure 4.16):

- The language allows one to declare constituents in a field-like manner by a name and the component interface they implement as their type.
- Interconnections between services of the constituents can be described by a simple statement which only needs to know the names of the two constituents involved as well as the names of the required and provided services to be connected.
- For interconnections between plugs, instead of the service names, the plug names are sufficient in most cases. The UCM runtime system which is able to interpret these composition descriptions determines a mapping between fitting services of both plugs to be connected and establishes the connections on service level. For the purpose of determining fitting services of both plugs, the algorithms described in Section 4.11 can be used. If these algorithms are not able to determine an unambiguous mapping, the connection information for the two plugs must include the service mapping explicitly.
- By a simple export statement, the services and plugs of the constituents which are exposed as the services and plugs of the composite component can be specified.
- In addition to the simple export statement for plugs which allows us to link a plug of a constituent to a compatible plug of the same size of the enclosing composite component, the composition language also supports plugs and single services of several constituents to be composed to a greater plug of the enclosing composite component. For this purpose, a more elaborate variant of the simple export statement exists.
- The composition language allows strict interface based programming. Constituents can be specified by their names and component interfaces only. Although one can specify the component to be used to instantiate this constituent by a certain binding statement, one is not forced to do so. If a binding is missing, the UCM runtime system searches for a fitting component by itself as described in Section 4.5. Strict interface based programming simplifies the exchange of components. New components can replace old ones without affecting existing compositions as far as they implement the same component interface. Even components of subtypes can replace old ones.

The next step in simplifying the composition process is to provide an appropriate tool support which even hides the composition language.

Tool Support: We contributed to tool support by our prototypes, the BPCE (see Section 5.1) and the CC-Builder (see Section 5.2). Whereas the CC-Builder focuses on hierarchical composition, the BPCE focuses on visual support for connections based on services and plugs and on simplifying the composition process especially by supporting the selection of suitable components and consistency checking of new compositions. Only some of the features of the BPCE, especially providing visual support, are summarized below:

- The BPCE allows one to connect component instances via services and plugs visually.
- On demand, the set of service providers fitting to a required service or plug selected for connection is depicted by displaying all fitting service providers in the composition window in a certain color.
- On demand, all component instances in the composition window which still have unresolved mandatory required services are highlighted by a certain coloring.
- A feature called *Guide connection* guides a user in establishing all needed connections in a predefined order.

When checking new compositions for consistency, the BPCE refers to our UCM type system described in Section 4.3.

In addition to these consistency checks, the type system allows us to decide, whether a component can be replaced by another one without invalidating any existing composition already referring to the component to be replaced.

Type System: In contrast to other type systems, the type definition of an UCM-component as well as the subtype relation between UCM-components distinguish optional and mandatory required services and include

- the connection point types belonging to required services as well as the lower and upper limit on the number of connections,
- the plugs declared,
- and our alias-constraints.

Adapters: A last means for simplifying compositions considered in this thesis are special adapters for multiplexing and delegation as described in Section 4.7.

7.2 Perspectives

Although we have already made a big step towards unifying and simplifying composition by providing fundamental concepts as well as concrete language and tool support, the following different perspectives demonstrate that our work can be extended in various directions which may be of interest for different groups of scientist as well as for industry. The perspectives are organized with respect to the main features of an ideal tool as described in Section 1.1.

Tool Box and Composition Window: It should be possible to easily import components not already prepared as UCM-components such that these components can be inserted into the tool box and used for composition. The BPCE already allows JavaBeans to be inserted into the tool box which are not aware of our component model. For such JavaBeans, the BPCE creates a component interface specification automatically. Such an import of unaware components should be extended to components of other industrial component models than JavaBeans and should include the automatic generation of the corresponding UCM implementation description.

Selection of Suitable Components: The determination and visualization of sets of fitting service providers as available by the BPCE should be extended to components in the tool box. This helps, if no fitting service provider resides in the composition window and a component from the tool box has to be selected prior to connection. A more advanced feature would allow us to specify a *best fitting* service provider with respect to certain criteria which are evaluated by our tool. Then an explicit selection of a suitable service provider can be avoided as far as the definition of a best fitting service provider allows this service provider to be determined unambiguously. Another useful feature would enable a user to specify different search criteria for components which are evaluated by the tool. Last but not least, in addition to pure syntactical aspects we should consider semantical aspects, too, when looking for fitting components. This would cause our type system to respect semantical aspects, too.

Visual Support for Connections: A great step to further simplify visual composition would be to allow a user to select a whole set of component instances for which he can demand that all needed connections are established automatically without further user interaction.

Hierarchical Composition: As our composition language, used to describe composite UCM-components, can be used to develop composite UCM-components directly that is, without the aid of a tool, it should be made more readable e.g., by using suitable keywords for connections, exports and implementation bindings.

A more elaborate task would be to allow composite UCM-components to be built from atomic UCM-components belonging to different industrial component models. To

reach this goal, a lot of work has still to be done in order to bridge the gap between the different representations of data types, the different infrastructures, the different reflection services used to instantiate components and to execute their methods etc.

Creation of New Applications: The CC-Builder allows a limited kind of application to be built. An application is essentially a composite UCM-component without any exports. In future releases it should be possible to enrich applications with initialization code which is executed at application start up.

Deployment: The deployment process realized so far supports storing of newly created components in the directories expected by the CC-Builder. In future releases we want to realize an implementation registry and a substitution registry as described in Section 4.5.

Reconfiguration: Based on our subtype relation between component interfaces, the UCM type system already allows us to decide whether a component can be substituted by another one. In the future we would like to support the whole substitution process by tools. The old component has to be removed and the new one deployed. The support should include the automatic creation of wrappers for the new components, if they implement component interfaces only related by a weak subtype relationship as described in Section 4.8.

Another reconfiguration task to be supported in future releases is the modification of existing composite UCM-components. At the moment, such composite components can be modified by changing their composition code manually. Future tool support should allow changes to be done visually. The tool should control all changes to the component implementation and component interface and allow a user only to extend the existing component interface to a compatible one with respect to our subtype relation.

Adaptation: In Section 4.7 we already introduced special adapters for multiplexing and delegation. Adaption can be further supported. If a component A should be replaced by a component B and B comes with incompatible service interface or connection point types, adaptation can be used to adapt the incompatible service interface or connection point types such that the adapted component can be used to replace A . Adaptations which only require to replace names, as e.g. method- and parameter names, could be supported by means of refactoring. Adapter chains as introduced in [Gsc02] can be used to finally adapt two types for which no single adapter is available.

7.3 Conclusion

Component based software composition is a dream almost as old as the idea of software engineering itself. Despite the wide recognition of the importance of this subject, the most recognized and disseminated component models today are commercial, influenced by little, if any, theoretical insights. Given this fact, it should come as no surprise that component models differ more than warranted by the posed problem (which is basically the same across all domains and technologies), and that uniform tool support is largely lacking.


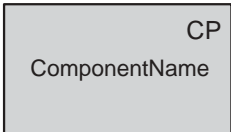
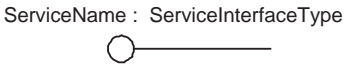
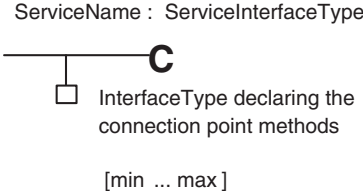
This thesis hopes to provide some theoretical underpinnings of several commercial component models and to transform them into a Unifying Component Model (UCM) that provides for a general framework for component composition, as well as for a conceptual basis for uniform tool support. A prototype tool based on this model has been presented.

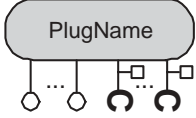
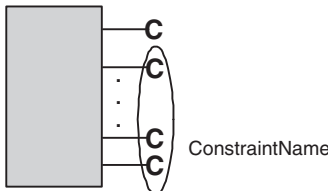
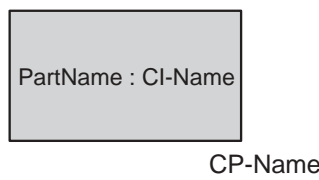
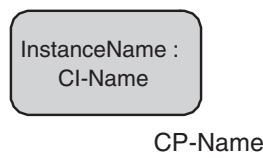

Appendix A

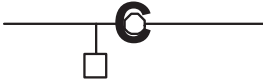

Summary of Used Graphical Elements

In the following two sections we list the graphical elements introduced in Chapter 4 and demonstrate their usage by typical diagrams.

A.1 Graphical Elements

Concept	Graphical Representation	Notes
Component Interface		See e.g. Section 4.
Component		CI-Name denotes the name of the component interface implemented by the component. See e.g. Section 4.
Provided Service		See e.g. Section 4.1.1.
Required Service		See e.g. Section 4.1.1.

Concept	Graphical Representation	Notes
Plug		See e.g. Section 4.1.1.
Constraint <i>Different Service Providers</i>		See e.g. Sections 4.1.1.3 and 4.3.2.3.
Part		CI-Name denotes the name of the component interface which types the part of a composite UCM-component <i>CC</i> . CP-Name denotes the name of the component implementation bound to the part or to the component interface it is typed by. The binding is done in the 'ImplementationBinding' section of <i>CC</i> . If no implementation binding exists for this part or for CI-Name, CP-Name is omitted. See e.g. Section 4.2.2.
Component Instance		CP-Name denotes the name of the component used to create the component instance. CI-Name denotes the component interface the component instance conforms to.
Link / Export		Denotes a link from a service of an internal part of a composite UCM-component to a service of its component interface. See e.g. Section 4.2.2, especially Figure 4.14.

Concept	Graphical Representation	Notes
Connections on Service Level		Denotes a connection between a provided service and a required service.
Implementation Relationship		Denotes that the component at the lower side of the arrow implements the component interface at the arrowhead.

Annotations like the names of components, component interfaces, part names, service interface types etc. or graphical representations of services may be omitted, if they are not of interest for what should be stressed in a special diagram.

A.2 Typical Diagrams

In the following some typical diagrams are shown using the graphical elements described in the previous section. Some of them were already shown in previous sections.

The following diagram derived from Figure 4.10 shows a typical graphical representation of a component interface with its provided and required services as well as its plugs.

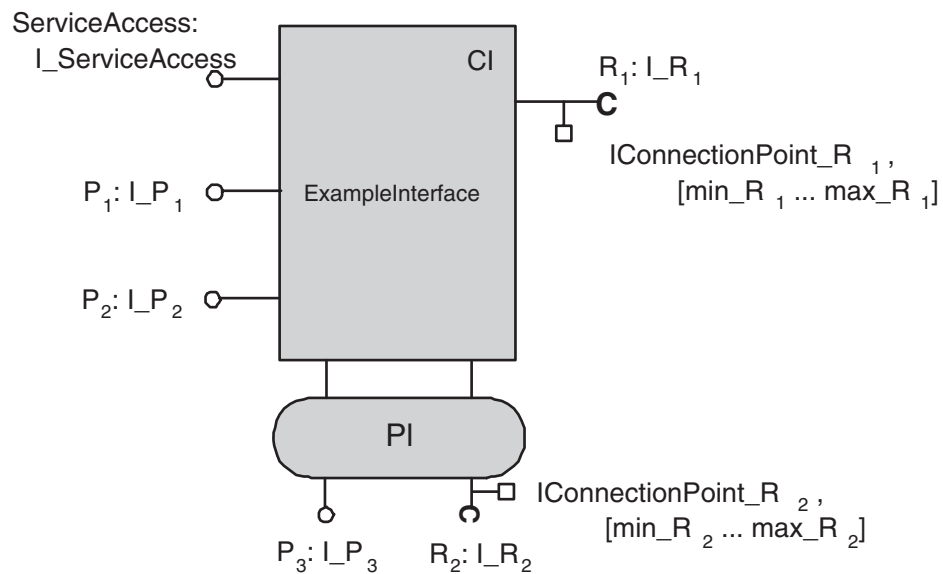


Figure A.1: Representation of a Component Interface

The next diagram shows a composite UCM-component with its aggregated parts, connections between the parts and links to the entities (services / plugs) of the component interface. The component interface is implicitly represented by the border surrounding the component and the services and plugs sticking out of this border.

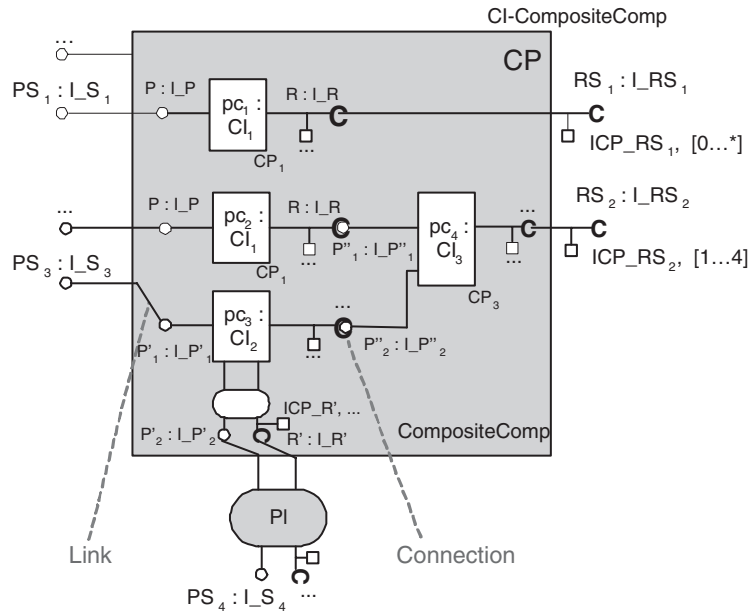


Figure A.2: Representation of a Composite UCM-Component

The following figure depicts a set of component instances where some of them are interconnected. These component instances together with their interconnections are created, if an instance of the composite UCM-component from Figure A.2 is created.

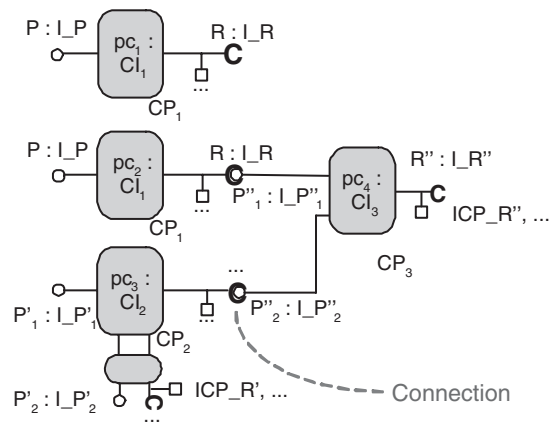


Figure A.3: Set of Component Instances

Bibliography

- [ACN01] J. Aldrich, C. Chambers, and D. Notkin. Component-oriented programming in ArchJava. In *OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Technical Report NU-CCS-01-06, pages 1 – 8. Northeastern University, Boston, MA, 2001.
- [ACN02] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. Technical Report Technical Report UW-CSE-02-04-01, University of Washington, 2002.
- [All97] R. Allen. A Formal Approach to Software Architecture. Ph.D Thesis CMU Technical Report CMU-CS-97-144, Carnegie Mellon University, 1997.
- [AN01] F. Achermann and O. Nierstrasz. Applications = Components + Scripts - A Tour of Piccola. In M. Aksit, editor, *Software Architectures and Component Technology*, pages 261 – 292. Kluwer, 2001.
- [Arc] ArchJava Language Reference Manual. Available at <http://archjava.fluid.cs.cmu.edu/papers/archjava-language.pdf>.
- [Bea] The Bean Builder. Available at <https://bean-builder.dev.java.net/>.
- [BH00] D. Birngruber and M. Hof. Interactive Decision Spots for JavaBeans Composition. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP 2000)*, June 2000. ISSN: 1103-1581.
- [Bir01] D. Birngruber. A Software Composition Language and Its Implementation. In D. Bjorner, M. Broy, and A. V. Zamlin, editors, *Perspectives of System Informatics (PSI 2001)*, volume 2244 of *Lecture Notes in Computer Science*, pages 519 – 529. Springer Verlag, 2001. Available at <http://www.ssw.unilinz.ac.at/General/Staff/DB/Research/Publications/>.
- [BM97] J. Bosch and S. Mitchell, editors. *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1375 of *Lecture Notes in Computer Science*. Springer Verlag, Juni 1997.

- [Bos97] J. Bosch. Adapting Object-Oriented Components. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 13 – 21. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
- [Bra01] P. Brada. Towards Automated Component Compatibility Assessment. In *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP 2001)*, 2001.
- [Bro97] D. Brookshier. *Java Beans Developer's Reference*. New Riders Publishing, Indianapolis, 1997.
- [BRS⁺00] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, New York, NY, 2000.
- [BS97] M. Buechi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 23 – 32. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
- [CER03] A. Charfi, D. Emsellem, and M. Riveill. Dynamic Component Composition in .NET. *Journal of Object Technology*, 3(2):37–46, 2003. Special issue: .NET: The Programmers Perspective: ECOOP Workshop 2003.
- [COM95] The Component Object Model Specification, Version 0.9, October 24 1995. Available at <http://www.microsoft.com/com/resources/comdocs.asp>.
- [Com01a] Component Pascal Language Report, March 2001. Available at <http://www.oberon.ch/pdf/CP-Lang.pdf>.
- [Com01b] What's New in Component Pascal?, March 2001. Available at <http://www.oberon.ch/pdf/CP-New.pdf>.
- [COR02] CORBA Components Version 3.0, June 2002. Document: formal/02-06-65, Available at <http://www.omg.org/technology/documents/formal/components.htm>.
- [CRN03] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment-time confinement checking. In *ACM Conference on*

- Object-Oriented Programming Systems, Languages and Applications (OOP-SLA'03)*, volume 38 of *ACM SIGPLAN Notices*, pages 374 – 387. ACM Press, October 2003.
- [DCO96] DCOM Technical Overview, November 1996. Available at <http://www.icons.com/support/pdfs/whitepapers/msdcom.pdf>, Last access: October 2006.
- [DCO98] Distributed Component Object Model Protocol-DCOM/1.0, draft, January 1998. Available at <http://samba.osmirror.nl/samba/ftp/specs/draft-brown-dcom-v1-spec-03.txt>, Last access: October 2006.
- [DP00] S. Denninger and I. Peters. *Enterprise JavaBeans*. Addison-Wesley, München, 2000.
- [EE98] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press Deutschland, Unterschleissheim, 1998. ISBN: 3-86063-459-3.
- [EJB03] Enterprise JavaBeans TM Specification, Version 2.1, November 12 2003. Available at <http://java.sun.com/products/ejb/docs.html#specs>.
- [Fow04] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2004.
- [FWA⁺99] N. Ford, E. Weber, T. Azzouka, T. Dietzler, J. Streeter, and C. Williams. *Borland JBuilder 3 Unleashed*. SAMS Publishing, 1999. ISBN: 0-672-31548-3.
- [GJS] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#289905.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *Java Language Specification*. Sun Microsystems, 1996.
- [GMW00] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
- [Gri98] F. Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt - Verlag, Heidelberg, 1998. ISBN: 3-932588-02-9.
- [Gro02] M. Groth. Creating Components in .NET, February 2002. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/componentsnet.asp>.

- [Gru99] J. Grundy. Aspect-Oriented Requirements Engineering for Component-Based Software Systems. In *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 84–91. IEEE CS Press, June 1999. ISBN 0-7695-0188-5.
- [Gsc02] T. Gschwind. *Adaptation And Composition Techniques for Component-Based Software Engineering*. PhD thesis, Vienna University of Technology, 2002.
- [GT00] V. Gruhn and A. Thiel. *Komponentenmodelle. DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Addison-Wesley, München, 2000.
- [HC98a] C. S. Horstmann and G. Cornell. *Core Java 1.1, Volume I - Fundamentals*. Sun Microsystems Press, Palo Alto, 1998. ISBN: 0-13-766965-8.
- [HC98b] C. S. Horstmann and G. Cornell. *Core Java 1.1, Volume II - Advanced Features*. Sun Microsystems Press, Palo Alto, 1998. ISBN: 0-13-766965-8.
- [HLS97] K. De Hondt, C. Lucas, and P. Steyaert. Reuse Contracts as Component Interface Descriptions. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 43 – 49. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. Available at <http://www.usingcsp.com/cspbook.pdf>.
- [HR99] D. Heuzeroth and R. Reussner. Dynamic coupling of binary components and its technical support. In *Proc. GCSE'99 Young Researchers Workshop, Erfurt*, pages 30–31, 1999.
- [Jav97] JavaBeans TM, Version 1.01, July 24 1997. Available at <http://java.sun.com/products/javabeans/docs/spec.html>.
- [JRH⁺04] M. Jeckle, C. Rupp, J. Hahn, B. Zengler, and S. Queins. *UML 2 glasklar*. Carl Hanser Verlag, 2004.
- [KR01] G. Kotonya and A. Rashid. A Development Strategy for Minimising Risks in Component-Based Development. In *Proceedings of the 27th Euro-micro Conference: Workshop on Component-Based Software Engineering, RE'99*, pages 12–21. IEEE Computer Society Press, September 2001.
- [LC99] M. Larsson and I. Crnkovic. New Challenges for Configuration Management. In *Proceedings of the SCM-9 workshop*, volume 1675 of LNCS, 1999.

- [LLF98] M. Leventhal, D. Lewis, and M. Fuchs. *Designing XML Internet Applications*. Prentice Hall PTR, Upper Saddle River, 1998. ISBN: 0-13-616822-1.
- [Löw05] J. Löwy. *Programming .NET Components*. O'Reilly, 2005. ISBN: 0-596-10207-0.
- [LR01] C. Lüer and D. S. Rosenblum. WREN - An Environment for Component-Based Development. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 207–217. ACM Press, 2001.
- [LvdH02] C. Lüer and A. van der Hoek. Composition Environments for Deployable Software Components. Technical Report UCI-ICS-02-18, Department of Information and Computer Science, University of California, Irvine, August 2002.
- [McI68] M. D. McIlroy. Mass-Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, pages 138–155, 1968.
- [MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153. Springer-Verlag, 1995.
- [MDK94] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):304–312, September 1994.
- [Mil91] R. Milner. The Polyadic pi Calculus: A tutorial. Technical Report ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, 1991.
- [MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, October 1996.
- [Mös98] H. Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer, Berlin, 1998. ISBN:3540646493.
- [MS97] L. Mikhajlov and E. Sekerinski. The Fragile Base Class Problem and Its Impact on Component Systems. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1375 of *Lecture Notes in Computer Science*, pages 353 – 363. Springer Verlag, Juni 1997. ISBN: 3-540-64039-8 ISSN: 0302-9743.

- [MSD] Grundlagen der Komponentenprogrammierung. <http://msdn.microsoft.com/library/deu/default.asp?url=/library/DEU/cpguide/html/cpconComponentProgrammingEssentials.asp>.
- [MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*, 26(1):70–93, December 2000. Available at <http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/trans/ts/&toc=comp/trans/ts/2000/01/e1toc.xml>.
- [Nie99] O. Nierstrasz. Piccola - A Small Composition Language. In *Proceedings of the Workshop on Object-Oriented Technology*, London, UK, 1999. Springer-Verlag.
- [NL97] O. Nierstrasz and M. Lumpe. Komponenten, Komponentenframeworks und Glueing. *HMD - Theorie und Praxis der Wirtschaftsinformatik*, 197:8 – 23, September 1997. ISBN: 3-89864-101-5 ISSN: 1436-3011.
- [Now99] P. Nowack. Interacting Components - A Conceptual Architecture Model. In *Proceedings of the Workshop on Object-Oriented Technology*, volume 1743 of *Lecture Notes in Computer Science*, pages 66–67, London, UK, 1999. Springer-Verlag.
- [NT95] O. Nierstrasz and D. Tsichritzis. *Object-oriented software composition*. Prentice Hall, 1995. ISBN: 0-13-220674-9.
- [Obe01] J. Oberleitner. The Component Workbench. Master’s thesis, Vienna University of Technology, October 2001.
- [OG02] J. Oberleitner and T. Gschwind. Composing Distributed Components with the Component Workbench. Technical Report TUV-1841-02-17, Vienna University of Technology, February 2002.
- [Pat00] T. Pattison. *Programming Distributed Applications with COM & Microsoft Visual Basic*. Microsoft Press, 2000.
- [Per87] D. E. Perry. Version Control in the Inscape Environment. In *Proceedings of ICSE87, Monterey, CA*, pages 142–149. ACM Press, 1987. ISBN 0-89791-216-0.
- [Pet95] M. T. Peterson. *DCE: A Guide to Developing Portable Applications*. McGraw-Hill, 1995.
- [PH02] A. Poetzsch-Heffter. Software-Architektur, 2002. Lecture at the University of Hagen.

- [Puc02] R. Pucella. Towards a Formalization for COM Part I: The Primitive Calculus. In C. Norris, B. James, and J. Fenwick, editors, *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 37 of *ACM SIGPLAN Notices*, pages 331 – 342. ACM Press, November 2002. ISBN:1-58113-471-1 ISSN: 0362-1340.
- [RJB05] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison Wesley, 2005.
- [RS02] R. Rinat and S. F. Smith. Modular Internet Programming with Cells. In Boris Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 257 – 280. Springer Verlag, 2002. ISBN: 3-540-43759-2.
- [SC00a] J. C. Seco and L. Caires. A Basic Model of Typed Components. In *ECOOP 2000 - Object-Oriented Programming*, volume 1850 of *LNCS*, pages 108 – 128. Springer Verlag, 2000.
- [SC00b] J. C. Seco and L. Caires. Parametrically Typed Components. In *WCOP 2000 - Workshop on Component Oriented Programming*, 2000.
- [SC02] J. C. Seco and L. Caires. ComponentJ: The Reference Manual. Technical Report UNL-DI-6-2002, Departamento de Informática FCT/UNL, 2002.
- [Sch97] S. Schreyjak. Coupling of Workflow and Component-Oriented Systems. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, TUCS General Publications No 5, pages 77 – 85. Turku Centre for Computer Science, September 1997. ISBN: 952-12-0039-1 ISSN: 1239-1905.
- [Sch05] U. Scheben. Hierarchical composition of industrial components. *Science of Computer Programming*, 56(1-2):117–139, April 2005. Available at <http://authors.elsevier.com/sd/article/S0167642304001807>.
- [SG99] J. P. Sousa and D. Garlan. Formal Modeling of the Enterprise JavaBeans Component Integration Framework. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 Formal Methods, World Congress on Formal Methods in the Development of Software Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 1281 – 1300. Springer Verlag, 1999.
- [SG01] J. P. Sousa and D. Garlan. Formal Modeling of the Enterprise JavaBeans Component Integration Framework. *Information and Software Technology*, 43(3):171 – 188, March 2001.

- [SGM03] F. Steimann, J. Gößner, and T. Mück. On the Key Role of Composition in Object-Oriented Modelling. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003: Proceedings of the 6th International Conference*, pages 106–120. Springer-Verlag, 2003.
- [Sie00] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, Inc., New York, 2000. ISBN: 0-471-29518-3.
- [SM05] F. Steimann and P. Mayer. Patterns of Interface-Based Programming. *Journal of Object Technology*, 4(5):75–94, 2005.
- [SPH01] U. Scheben and A. Poetzsch-Heffter. Demonstration der Problematiken beim Einsatz von Komponenten anhand eines Lernkurses aus JavaBeans-Komponenten. Technical Report AIB-2001-11, RWTH Aachen, Department of Computer Science, December 2001. ISSN: 0935-3232.
- [SPH03] U. Scheben and A. Poetzsch-Heffter. Concepts and Techniques simplifying the Assembly Process for Component Instances. In U. Assmann, E. Pulvermüller, I. Borne, N. Bouraqadi, and P. Cointe, editors, *SC 2003: Workshop on Software Composition Affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier, April 2003. Available at <http://www.elsevier.nl/jeing/31/29/23/133/47/33/82.5.011.pdf>.
- [SPJF02] A. Speck, E. Pulvermüller, M. Jerger, and B. Franczyk. Component Composition Validation. *International Journal of Applied Mathematics and Computer Science*, 12(4):581 – 589, December 2002.
- [SR02] H. Schmidt and R. Reussner. Automatic component adaption by concurrent state machine retrofitting. Technical Report 2000/81, School of Computer Science and Software Engineering, Monash University, Melbourne, 2002.
- [Sre01] V.C. Sreedhar. ACOEL: A component-oriented extensional language. Technical report, IBM T.J. Watson Research Center, 2001.
- [Str00] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Boston, 2000. ISBN: 0-201-70073-5.
- [Szy98] C. Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1998.
- [TW01] P. Tabatt and H. Wolf. *Java programmieren mit JBuilder4*. Software & Support Verlag GmbH, Frankfurt, 2001. ISBN: 3-935042-04-3.

- [UML05] Unified Modeling Language: Superstructure version 2.0, August 2005. Available at <http://www.omg.org/cgi-bin/doc?formal/05-07-04>.
- [vdAvHvdT02] W. M. P. van der Aalst, K. M. van Hee, and R. A. van der Toorn. Component-based software architectures: A framework based on inheritance of behavior. *Science of Computer Programming*, 42(2-3):129–171, 2002.
- [Völ03] M. Völter. A Taxonomy of Components. *Journal of Object Technology*, 2(4):119–125, July-August 2003.
- [WCD⁺01] S. Weerawarana, F. Curbera, M. J. Duftler, D. A. Epstein, and J. Kesselman. Bean Markup Language: A Composition Language for JavaBeans Components. In *6th USENIX Conference on Object-Oriented Technologies and Systems, January 29 - February 2, 2001, San Antonio, Texas, USA*, COOTS. USENIX, 2001. Available at <http://www.usenix.org/publications/library/proceedings/coots01/weerawarana.html>.
- [Wes02] R. Westphal. *.NET kompakt*. Spektrum Akademischer Verlag, Heidelberg; Berlin, 2002. ISBN: 3-8274-1185-8.
- [WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
- [Zen02] M. Zenger. Type-Safe Prototype-Based Component Evolution. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 470 – 497. Springer Verlag, 2002.
- [ZW97] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

Index

- .NET, 49
 - assembly, 50, 53
 - attributes, 50
 - common language runtime, 50
 - common type system, 50
 - component, 53
 - component interface, 54
 - component lookup, 58
 - composition techniques, 60
 - events, 55
 - interface implementation
 - explicit, 50
 - implicit, 50
 - intermediate language, 50
 - properties, 54
 - reflection, 65
 - substitutability, 67
 - type metadata, 65
 - type system, 64
- abbreviations
 - ADL, 251
 - BDK, 18
 - CBD, 2
 - CCM, 2
 - CLR, 50
 - CLSID, 33
 - CO, 132
 - COM, 2
 - Constraints (*CI*), 132
 - CTS, 50
 - EJB, 2
 - IDL, 31
 - IID, 31
 - IL, 50
 - JAR file, 18
 - MIDL, 31
 - PL, 132
 - Plugs (*CI*), 132
 - Provided (*CI*), 132
 - Provided (*CI, Pl*), 132
 - PS, 132
 - Required (*CI*), 132
 - Required (*CI, Pl*), 132
 - RS, 132
 - ser-file, 18
 - UCM, 85
 - UML, 259
- adapter, 24
- architecture description language, 251
 - Acme, 253
 - Darwin, 251
 - Wright, 256
- assembly, 13, 73, 75, 81, 92, 117, 145, 149, 151, 228, 237, 247
- assembly time, 14
- assembly tool, 13
- client, 14
- COM, 30, 69, 71, 74, 77, 80, 122
 - aggregation, 38, 118
 - category, 43
 - class factory, 30
 - class identifier (CLSID), 33
 - coclass-declaration, 33
 - COM class, 30, 269
 - COM object, 30
 - component, 30, 178
 - component interface, 31
 - component lookup, 33
 - composition techniques, 37
 - connectable object, 31, 37

- connection point object, 37
- emulation, 49
- IConnectionPoint, 37
- IConnectionPointContainer, 37
- IDL compiler, 31
- integration, 190, 211
- interface, 30, 133
- interface definition language, 31
- interface identifier (IID), 31
- interface pointer, 34
- IUnknown interface, 31
- outgoing interface, 31, 37
- provided interface, 31
- QueryInterface, 33
- server, 30, 269
 - in-process server, 30
 - local server, 30
 - remote server, 30
- sink object, 37
- substitutability, 48
- type COM class, 42
- type COM interface, 40
- type library, 44
 - COM class:type description, 46
 - COM interface:type description, 47
 - ITypeInfo, 45
 - ITypeLib, 44
 - typeinfo, 44
- type metadata, 44
- type system, 40
- valid composition, 48
- virtual function table, 35
- complementary plugs, 155
- component, 12, 110, 256, 259
- component instance, 12
- component interface, 12, 110
- component model, 13, 15, 68, 73, 246
 - .NET, 49
 - COM, 30
 - JavaBeans, 17
 - UCM, 2, 68, 82, 85, 275
- component-oriented language, 234
 - ACOEL, 244
 - ArchJava, 235
 - Cells, 245
 - Component Pascal, 240
 - ComponentJ, 237
 - prototype based component evolution, 244
- composite, 14
- composite component, 14
- composite structure, 259, 274
- composition, 14
- composition language, 14, 73, 234, 245, 277
 - Bean Plans, 247
 - BML, 246
 - Piccola, 249
- composition techniques
 - .NET, 60
 - COM, 37
 - JavaBeans, 22
 - UCM, 118
- configuration time, 14
- connection, 13, 91
 - via events
 - .NET, 55
 - JavaBeans, 23
 - via outgoing interfaces, 37
 - via plugs, 120
 - via services, 119
- connection point interface, 92
- connection point object, 37, 91
- connector, 245, 251, 253, 256, 263
 - assembly, 264
 - delegation, 264
- constituent, 14, 83, 84, 119
- constraint 'Different Service Provider', 108
- dependency
 - dynamic, 15
 - static, 15
- deployment, 14
- enabling required service interface, 13
- examples

- ambiguous mapping, 221
- automatically resolving ambiguity, 219
- check for complementary plugs, 217
- collection customer data, 86
- component implementation, 124
- component instances, service access, interconnections, 93
- component interface for data logging, 113
- component interface of model, 115
- component interface type with constraints, 137
- component interface type with plugs, 137
- component interface with constraints, 116
- component interfaces for data logging, 143
- compound documents, 104
- connect-method specification, 127
- declarations in IDL, 32
- default connect-method, 118
- event handling JavaBeans, 20
- flexible customer form, 106
- model view controller, 102
- packaging machines, 100
- subtyping of component interfaces, 150
- subtyping of plugs, 144
- UCM descriptions for .NET, 202
- wiring of beans based on events, 23
- wordprocessor, 72
- wordprocessor connections, 89
- fully linked plugs, 160
- ImplementationBinding section, 123, 164, 174, 175, 270
- implicit interface, 15, 54, 59, 77, 237
- JavaBeans, 17
 - adapter, 24
 - BeanInfo, 21, 25, 27
 - component, 18
 - component interface, 19
 - component lookup, 21
 - composition techniques, 22
 - event listener, 19
 - event source, 19
 - events, 19
 - EventSetDescriptor, 27
 - introspector, 21, 28
 - Java archive, 18
 - manifest file, 18
 - MethodDescriptor, 27
 - patterns
 - (de)registration methods, 20
 - notification methods, 19
 - properties, 19
 - property editor, 21
 - PropertyDescriptor, 27
 - reflection, 26
 - serialized bean, 18
 - substitutability, 29
 - type metadata, 26
 - type system, 25
 - wiring of beans, 23, 55
- link, 266
 - fully linked plugs, 160
 - linked plugs, 160
 - linked provided services, 157
 - linked required services, 158
 - partially linked plugs, 160
- linked plugs, 160
- linked provided services, 157
- linked required services, 158
- matching services, 154
- object implementing a method, 153
- object implementing an interface, 153
- part, 11, 83, 121, 231, 232, 236, 237, 244, 247, 263
- partially linked plugs, 160
- plug, 99
- port, 235, 237, 253, 256, 264

- provided service, 88
- proxy, 90
- required service, 88
 - mandatory, 89
 - optional, 89
- server, 14
- service, 88
- service 'ServiceAccess', 92
- service interface, 12
- service object, 89
- service provider, 89
- substitutability
 - .NET, 67
 - COM, 48
 - JavaBeans, 29
 - UCM, 166
- subtype definitions
 - for .NET components, 64
 - for COM classes, 44
 - for COM classes (Szyperski), 43
 - for COM interfaces, 41
 - for JavaBeans, 26
 - UCM-components
 - for cardinality types, 139
 - for component interfaces, 145
 - for method based interface types, 138
 - for plugs, 141
 - for provided services, 138
 - for required services, 139
- suitable service provider, 89
- terms
 - complementary plugs, 155
 - component, 110
 - component interface, 110
 - connection, 91
 - connection point interface, 92
 - connection point object, 91
 - constraint 'Different Service Provider', 108
 - fully linked plugs, 160
 - link between provided services, 157
 - link between required services, 158
 - linked plugs, 160
 - matching services, 154
 - object implementing a method, 153
 - object implementing an interface, 153
 - partially linked plugs, 160
 - plug, 99
 - provided service, 88
 - proxy, 90
 - required service, 88
 - mandatory, 89
 - optional, 89
 - service, 88
 - service 'ServiceAccess', 92
 - service object, 89
 - suitable service provider, 89
- theorems
 - compatible components, 175
 - polymorphic component instances, 168
 - transitivity of subtyping for interfaces, 138
 - transitivity of subtyping for plugs, 142
 - transitivity of subtyping for required services, 139
 - valid export plugs, 160
 - valid export provided services, 158
 - valid export required services, 158
 - valid plug composition, 161
 - valid plug connection, 156
 - valid service connection, 154
- type definitions
 - .NET component type, 64
 - COM class type, 43
 - COM class type (Szyperski), 42
 - COM interface type, 41
 - JavaBean type, 25
 - UCM-components
 - component interface type, 137
 - constraint type, 136
 - method based interface type, 133
 - method type, 133
 - plug type, 136

- provided service type, 134
 - required service type, 134
 - service interface type, 133
- type identifier
 - Cardtype, 134
 - CIttype, 137
 - CPtype, 134
 - Ctype, 136
 - Itype, 133
 - Mtype, 133
 - PLtype, 136
 - PStype, 134
 - RStype, 134
 - SIttype, 133
- type system, 77
 - .NET, 64, 78
 - CCM, 78
 - COM, 40, 77
 - EJB, 79
 - JavaBeans, 25, 77
 - UCM, 132
- UCM, 85
 - component implementation
 - atomic components, 122
 - composite components, 123
 - component interface, 110, 111
 - component lookup, 162
 - composition techniques, 118
 - correctness composition, 151
 - substitutability, 166
 - type system, 132
- unified modeling language, 259
- wiring, 13, 23

Verzeichnis der zuletzt erschienenen Informatik-Berichte

- [322] Abellanas, M., Hurtado, F., Icking, Chr., Ma, L., Palop, B., Ramos, P. A.:
Best Fitting Rectangles
- [323] Roth, J.:
A Decentralized Location Service Providing Semantic Locations
- [324] Roth, J.:
2. GI/ITG KuVS Fachgespräch Ortsbezogene Anwendungen und Dienste
- [325] Fernandez, A.:
Groupware for Collaborative Tailoring
- [326] Grubba, T., Hertling, P., Tsuiki, H., Weihrauch, K.:
CCA 2005 - Second International Conference on Computability and Complexity in Analysis
- [327] Heutelbeck, D.: Distributed Space Partitioning Trees and their Application in Mobile Computing
- [328] Widera, M., Messing, B., Kern-Isberner, G., Isberner, M., Beierle, C.:
Ein erweiterbares System für die Spezifikation und Generierung interaktiver Selbsttestaufgaben
- [329] Fechner, B.:
A Fault-Tolerant Dynamic Multithreaded Microprocessor
- [330] Keller, J., Schneeweiss, W.:
Computing Closed Solutions of Linear Recursions with Applications in Reliability Modelling
- [331] Keller, J.:
Efficient Sampling of the Structure of Cryptographic Generators' State Transition Graphs
- [332] Fisseler, J., Kern-Isberner, G., Koch, A., Müller, Chr., Beierle, Chr.:
CondorCKD – Implementing an Algebraic Knowledge Discovery System in a Functional Programming Language
- [333] Cenzer, D., Dillhage, R., Grubba, T., Weihrauch, K.:
CCA 2006 - Third International Conference on Computability and Complexity in Analysis
- [334] Fechner, B., Keller, J.:
Enhancement and Analysis of a Simple and Efficient VLSI Model
- [335] Wilkes, W., Ondracek, N., Oancea, M., Seiceanu, M.:
Web services to resolve concept identifiers supporting effective product data exchange
- [336] Kunze, C., Lemnitzer, L., Osswald, R. (eds.):
GLDV-2007 Workshop - Lexical-Semantic and Ontological Resources