Component-based Adaptation Methods for Service-Oriented Peer-to-Peer Software Architectures

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Sascha Alda
aus Königswinter

Bonn, 2006

Angefertigt mit der Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn. Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn http://hss.ulb.uni-bonn.de/diss_online elektronisch publiziert.

1. Referent: Univ.-Prof. Dr. Armin B. Cremers, Universität Bonn

2. Referent: Univ.-Prof. Dr. Jürgen Ebert, Universität Koblenz-Landau

Tag der Promotion: 21.12.2006

Erscheinungsjahr 2007

Abstract

Service-oriented peer-to-peer architectures aim at supporting application scenarios of dispersed collaborating groups in which the participating users are capable of providing and consuming local resources in terms of *peer services*. From a conceptual perspective, service-oriented peer-to-peer architectures adopt relevant concepts of two well-established state-of-the-art software architectural styles, namely service-oriented architectures (also known as SOA) and peer-to-peer architectures (P2P). One major argumentation of this thesis is that the adoption of end-user adaptability (or *tailorability*) concepts is of major importance for the successful deployment of service-oriented peer-to-peer architectures that support user collaboration. Since tailorability concepts have so far not been analyzed for both peer-to-peer and service-oriented architectures, no relevant models exist that could serve as a tailorability model for service-oriented peer-to-peer architectures.

In order to master the adaptation of peer services, as well as peer service compositions within service-oriented peer-to-peer architectures, this dissertation proposes the adoption of component-oriented development methods. These so-called *component-based adaptation* methods enable service providers to adapt their provided services during runtime. Here, a model for analyzing existing dependencies on *subscribed* service consumers ensures that a service provider is able to adapt his peer services without violating any dependencies. In doing so, an *adaptation policy* that can be prearranged within a peer group regulates the procedures of how to cope with existing dependencies in the scope of a group. The same methods also serve as a way to *handle exceptional cases*, in particular the failure of a dependent service provider peer and, hence, a service that is part of a local service composition. In this, the hosting runtime environment is responsible for *detecting exceptions* and for initiating the process of *exception resolution*. During the resolution phase, a user can be actively involved at selected decision points in order to resolve the occurred exception in unpredictable *contexts*. An exception could also be the reason for the violation of an *integrity constraint* that serves as a contract between various peers that interact within a given collaboration. The notion of integrity constraints and the model of handling the constraint violation aim at improving the *reliability* of target-oriented peer collaborations.

This dissertation is composed of three major parts that each makes a significant contribution to the state of the art. First of all, a formal architectural style ($SO_{P2P}A$) is introduced to define the fundamental elements that are necessary to build service-oriented peer-to-peer architectures, as well as their relationships, constraints, and operational semantics. This architectural style also formalizes the above-mentioned adaptation methods, the exception handling model that embraces these methods, the analysis model for managing consumer dependencies, as well as the integrity constraints model. Subsequently, on this formal basis, a concrete (specific) service-

oriented peer-to-peer architecture (DEEVOLVE) is conceptualized that serves as the default implementation of that style. Here, the notions described above are materialized based on state-of-the-art software engineering methods and models. Finally, the third contribution of this work outlines an application scenario stemming from the area of construction informatics, in which the default implementation DEEVOLVE is deployed in order to support dispersed planning activities of structural engineers.

Danksagung

Die vorliegende Doktorarbeit wurde während meiner Anstellung als wissenschaftlicher Mitarbeiter am Institut für Informatik III der Universität Bonn in dem Zeitraum 2001 bis Ende 2006 angefertigt. In dieser Zeit musste ich neben einigen beruflichen und privaten Erfolgen auch eine Reihe von privaten Rückschlägen, unter anderem den frühen Tod meiner Mutter sowie einen schweren Verkehrsunfall, erfahren. Viele der nachfolgend genannten Personen haben nicht nur zum Gelingen der vorliegenden Arbeit, sondern auch zur Überwindung der eben genannten Ereignisse beigetragen.

An erster Stelle möchte ich mich bei Herrn Prof. Dr. Armin B. Cremers für seine kompetente und fachliche Betreuung während der Erstellung dieser Arbeit recht herzlich bedanken. In vielen Diskussionen und Gesprächen konnte er durch richtungweisende Hinweise und Kommentare entscheidend zur Fertigstellung des Promotionsvorhabens beitragen. Generell versteht er es sehr gut, seinen Mitarbeitern eine produktive und liberale Arbeitsatmosphäre zu gewähren, von der ich in dieser Zeit ebenfalls sehr profitieren konnte. Des Weiteren möchte ich bei Herrn Prof. Dr. Jürgen Ebert für seine Zweitbetreuung der Arbeit bedanken. Er konnte mir vor allem bei der sehr aufwendigen und komplexen Darstellung des formalen Architekturstils (Kapitel 3 und 4) viele Anregungen geben, welche seine Verständlichkeit um ein Vielfaches verbesserte.

Eine Doktorarbeit sollte bekannterweise nicht nur bei den betreuenden Professoren, sondern auch bei den übrigen Kollegen und Doktoranden Gefallen finden. Ihr Urteil kann entscheidend zum Erfolg aber auch zum Misserfolg einer Dissertation beitragen. An dieser Stelle seien die Mitglieder des damaligen Fachbereichs ProSEC, vor allem aber Herrn Dr. Markus Won und Herrn Juniorprofessor Dr. Volkmar Pipek gedankt, die meine Ideen und Ansätze zu dieser Arbeit bereits in einer frühen Phase verstanden und unterstützt haben. Des Weiteren konnte ich von den regelmäßigen Zusammenkünften der Arbeitsgemeinschaft „AnonDr“ profitieren. In dieser Runde hatte ich die Gelegenheit, meine Arbeit einer Reihe von Kollegen vorzustellen. Ihren konstruktiven Anmerkungen sei Dank, dass die Arbeit in dieser Form vollendet werden konnte. Aus dieser Gruppe möchte ich mich vor allem bei Herrn Dr. Pascal Costanza, Frau Dr. Melanie Gnasa, Frau Julia Kuck, Herrn Dr. Uwe Radetzki, Herrn Tobias Rho, Herrn Daniel Speicher, Herrn Dr. Markus Won auch für die Zusammenarbeit über dieser Arbeitsgemeinschaft hinaus bedanken. Herrn Prof. Dr. Andreas Weber sei für seine Rolle als Drittgutachter sowie als Vorsitzender des Promotionsausschusses gedankt.

Es sei einer Reihe von weiteren, externen Leuten, die im Kontext für das Zustandekommen dieser Dissertation stehen, gedankt. Herrn Prof. Dr.-Ing. Hartmann (Ruhr-Universität Bochum) hat als Leiter der AG „Agenten“ im DFG-Schwerpunktprogramm 1103 sowie nicht zuletzt als Rolle des fachfremden Gutachters stets sehr viele inhaltliche Aspekte zur Darstellung der Anwendungsszenarien (Kapitel 9)

beitragen können. Aus dieser Arbeitsgemeinschaft sei auch die sehr enge und jahrelange Zusammenarbeit mit Herrn Dr.-Ing. Jochen Bilek sowie mit Herrn Dr.-Ing. Mirko Theiss erwähnt, die mit ihrer ingenieurgeprägten Sichtweise ebenfalls viele Ideen zu dieser Arbeit beitrugen. Einen besonderen Dank sei an Herrn Prof. Dr. Martin Fassnacht von der Wissenschaftlichen Hochschule der Unternehmensführung (WHU) in Koblenz gerichtet, bei dem ich als ehemalige studentische Hilfskraft zwei Jahre arbeiten durfte. Er hat mich mit seiner professionellen und ehrgeizigen Arbeitsweise wie kaum ein anderer in den drauf folgenden Jahren geprägt.

Dank gebührt auch den vielen Studierenden, die meine Arbeit in letzten Jahren als Diplomand/-in, als wissenschaftliche oder als studentische Hilfskraft unterstützt haben: Musan Ahmetavic, Markus Blätzinger, Gerwin Brill, Lukas Degener, Ali Reza Farnoudi, Jasmin Grigull, Eddin Idlbi, Hannes Korte, Todor Mitrov, Philippe Nuderscher, Aleksej „Aleks“ Palij, Florian Schmidt sowie Friedericke Wolfrum.

„Last but not least“ möchte ich mich bei meinen Freunden André, Andreas, Axel, Rolf, Thomas u.a. für all ihre Kraft und ihren *Spirit* in den letzten 15 Jahren, aber auch für ihr schier endloses Bestreben unsere Freundschaft weiterzuführen, bedanken. Ebenso gebühren natürlich meinem Vater sowie meinen Brüdern Ingo und Helmut sowie seiner Frau Anne einen mehr als lieben Dank für all ihre Unterstützung in den vergangenen Jahren.

Gewidmet sei diese Arbeit meiner Mutter († 2002).

Table of Contents

ABSTRACT V

DANKSAGUNG VII

TABLE OF CONTENTS IX

LIST OF FIGURES XV

LIST OF TABLES XIX

CHAPTER 1 INTRODUCTION AND MOTIVATION 1

1.1 Description of the Problem Area 1

1.2 Research Questions 4

1.3 Contributions 6

1.3.1 The $SO_{P2P}A$ Architectural Style 7

1.3.2 The DEEVOLVE Architecture 8

1.3.3 Evaluation of the DEEVOLVE Architecture in CoBE 9

1.4 Structure of this Dissertation 10

CHAPTER 2 SERVICE-ORIENTED PEER-TO-PEER ARCHITECTURES: CLASSIFICATION INTO THE STATE OF THE ART 11

2.1 Preliminaries 11

2.1.1 Software Architectures 12

2.1.2 Adaptability of Software 13

2.1.2.1 Adaptability 13

2.1.2.2 Tailorability 14

2.1.2.3 Adaptivity 15

2.2 Peer-to-Peer Architectures 16

2.2.1 Definition and Characteristics 16

2.2.2 Self-Organizing Peer-to-Peer Architectures 19

2.2.3 Reputation and Trust 20

2.2.4 Overview of existing Peer-to-Peer Systems 20

2.2.5 JXTA – A Standard Peer-to-Peer Framework 21

2.3 Service-Oriented Architectures (SOA) 22

2.3.1 Definition and Characteristics 22
2.3.2 Web Services 23
2.3.3 Service Composition 23
2.3.3.1 Choreography 24
2.3.3.2 Orchestration 24
2.3.3.3 Composition Languages for Web Services 25
2.3.4 Comparison: SOA vs. Peer-to-Peer 26
2.4 Discussion: Service-Oriented Peer-to-Peer Architectures 27
2.4.1 Starting Point: Requirements of a Distributed Groupware System 27
2.4.2 Discussion: Appropriateness of Peer-to-Peer and SOA 29
2.4.2.1 Aspect: Service Composition 29
2.4.2.2 Aspect: Exception Handling 31
2.4.2.3 Aspect: Adaptability 33
2.4.3 Suggestion: Service-oriented Peer-to-Peer Architecture 34
2.4.4 Concluding Remarks 35
2.5 Component Orientation 35
2.5.1 Definition and Characteristics 36
2.5.2 Component Models 37
2.5.3 Component Composition 39
2.5.3.1 Component-oriented vs. Service-oriented Composition 40
2.5.3.2 Exception Handling in Component-based Compositions 41
2.5.3.3 Implications for a Service-oriented Peer-to-Peer Architecture 43
2.5.4 Component-based Adaptation (Tailoring) 43
2.5.4.1 An Overview and Characteristics 43
2.5.4.2 Implications for a Service-oriented Peer-to-Peer Architecture 44
2.6 Conclusion 45
2.7 Next Steps 45
CHAPTER 3 FORMALIZATION OF THE ARCHITECTURAL STYLE FOR SERVICE-ORIENTED PEER-TO-PEER ARCHITECTURES ($SO_{P2P}A$) 47
3.1 The pi-calculus 47
3.2 Justification for Applying the pi-calculus 48
3.3 The Core Elements of $SO_{P2P}A$ 48
3.3.1 Principle Overview 49
3.3.2 Syntax of $SO_{P2P}A$ 52
3.3.3 Type System of $SO_{P2P}A$ 54
3.3.4 Operational Semantics of $SO_{P2P}A$ 59
3.3.5 Components 61
3.3.6 Component Composition 67
3.3.7 Peer Service 70
3.3.8 Peer 75

3.3.8.1 Peer-to-Peer Architecture ... 76
3.3.8.2 Routing Mechanisms for Rendezvous Peers ... 76
3.3.8.3 Message Exchange ... 77
3.3.9 Peer Group ... 78
3.3.9.1 Adaptation Policy ... 78
3.3.9.2 Applying, Joining, and Resigning a Group ... 79
3.3.10 The Deployment of a Peer Service ... 79
3.3.10.1 Discovering a Peer Service ... 80
3.3.10.2 Creating a Peer Service ... 81
3.3.10.3 Obtaining the Interface Composition Part of a Peer Service ... 82
3.3.10.4 Obtaining the Composition Agent for the Interface Composition 83
3.3.10.5 Obtaining the Default Handler ... 84
3.3.10.6 Subscription of a Consumer Peer for a Peer Service ... 84
3.3.11 The Invocation of a Peer Service ... 85
3.3.12 Composition of Peer Services ... 87
3.3.13 Execution Models ... 88
3.4 Summary ... 90
CHAPTER 4 COMPONENT-BASED ADAPTATION METHODS IN $SO_{P2P}A$ ... 91
4.1 The Adaptation of a Peer Service ... 91
4.1.1 Component-based Adaptation Methods (*Adaptationprocess*) ... 93
4.1.2 Consumer Dependency Analysis ... 96
4.1.3 Tailoring a published Peer Service ... 98
4.1.4 Tailoring an Interface of a Consumed Service ... 101
4.2 Integrity Constraints ... 102
4.2.1 Definition of Integrity Constraints ... 103
4.2.2 Evaluation of Integrity Constraints ... 104
4.3 Exception Handling ... 105
4.3.1 Exception Detection ... 106
4.3.2 Exception Resolution ... 108
4.3.3 Exception Cascading ... 108
4.4 Related Work and Scope ... 109
CHAPTER 5 ASSESSMENT OF FREEVOLVE'S CONCEPTS ... 115
5.1 Core concepts of FREEVOLVE ... 115
5.1.1 The FLEXIBEAN Component Model ... 115
5.1.2 Tailoring Components ... 117
5.1.3 Proxy Objects ... 118
5.1.4 Prototypical Implementation and Applications ... 120
5.2 Additional Concepts ... 120
5.2.1 Server Sessions ... 120
5.2.2 Semantic Integrity Conditions ... 121

5.3 Assessment of the presented concepts 122
5.3.1 Component Model and Component Composition 122
5.3.2 Component-based Tailorability 124
5.3.3 Deployment of Distributed Applications (Server Session) 125
5.3.4 Semantic Integrity Conditions (Exception Handling) 126
5.4 Conclusion and next Steps 127
CHAPTER 6 THE DEEVOLVE RUNTIME ENVIRONMENT 131
6.1 Overview of the Architecture 131
6.1.1 Relation to FREEVOLVE 131
6.1.2 JXTA as the fundamental Framework 132
6.2 Peer and Peer Groups 132
6.2.1 Peer 133
6.2.2 Peer-to-peer architecture 133
6.2.3 Peer Group 134
6.2.4 Advertisements 135
6.2.5 Modeling Peers and Peer Groups for concrete Applications 135
6.3 Peer Services 137
6.3.1 Component Model 137
6.3.2 Component Composition 138
6.3.3 Peer Service Model 138
6.3.4 Structural Model of a Peer Service 139
6.3.5 Modeling of Peer Services with CAT-XML 141
6.3.6 Advertisement of Peer Services 142
6.3.7 Peer Service Discovery and Publication 143
6.4 Peer Service Composition 144
6.4.1 Modeling a Peer Service Composition 144
6.4.2 Structural Model of Service Composition 144
6.4.3 PeerCAT Composition Language 145
6.5 Peer Service Execution Model 147
6.5.1 Single Service Execution 147
6.5.2 Basic Service Composition 148
6.5.3 Distributed Service Composition 149
6.6 Prototype Implementation of DEEVOLVE 150
6.6.1 Layered Architecture of a local Peer Environment 150
6.6.2 Starting the local DEEVOLVE Peer Environment 151
6.6.3 The DEEVOLVE Console 151
6.6.4 The DEEVOLVE Messaging Service 152
6.6.5 Composition Tools 153
6.7 Evaluation 154
6.8 Conclusion 154

CHAPTER 7 CONSUMER DEPENDENCY MANAGEMENT IN DEEVOLVE 155
7.1 Overview on the Concept 155
7.1.1 Extension to the Structural Model of DEEVOLVE 156
7.2 Subscription to a Peer Service 157
7.2.1 Process of Subscription 157
7.2.2 Storing Subscription Data 158
7.2.3 Dependency Values 160
7.2.4 Updating a Dependency Value and Unsubscription 161
7.3 Adaptation Policy 163
7.3.1 Adaptation Strategies 163
7.3.2 Adaptation Condition 164
7.3.3 Evaluation of an Adaptation Policy 166
7.3.4 Advertisement of an Adaptation Policy 167
7.4 Design and Prototypical Implementation 168
7.4.1 Visualizing Consumer Dependencies 168
7.4.2 Analyzing Consumer Dependencies 170
7.4.3 Executing the proposed Adaptation Strategy 171
7.4.4 Extensibility Mechanisms 172
7.5 Tailoring a local Peer Service 172
7.6 Related Work and Scope 173
CHAPTER 8 ADAPTATION METHODS FOR RUNTIME EXCEPTION HANDLING 175
8.1 Overview on Exception Handling in DEEVOLVE 175
8.2 Integrity Constraints 176
8.2.1 Principle Idea 177
8.2.2 Definition of Integrity Constraints in PeerCAT 177
8.2.3 Example 1: Minimal Composition Integrity 179
8.2.4 Example 2: Information Flow Integrity 181
8.2.5 The Deployment of Integrity Constraints 182
8.3 Exception Detection 183
8.3.1 Monitoring of Peers 183
8.3.2 Broker-based Remote Interaction 184
8.3.3 Evaluation of Broker Generation 186
8.3.4 Verifying Integrity Constraints 186
8.3.5 Exception Cascading 187
8.4 Exception Resolution 187
8.4.1 Definition of Exception Handlers in PeerCAT 188
8.4.2 Adaptation Methods as Actions for Handling Exceptions 189
8.4.3 Adaptive vs. End-User Exception Handling 191
8.4.4 Deployment and Execution of Exception Handlers 191

8.5 Handling Exceptions at various Levels of Complexity 194
8.6 Related Work 196
8.7 Summary 200
CHAPTER 9 EVALUATION OF DEEVOLVE IN THE COBE PROJECT 201
9.1 Characteristics of modern Projects in Construction Engineering 201
9.2 Goals of the COBE project 203
9.3 Result of the Requirements Analysis 204
9.4 Results from the COBE project 205
9.4.1 DEEVOLVE as the Solution for a Software platform 205
9.4.1.1 Principle Aspects 205
9.4.1.2 Bridge Components to integrate Legacy Applications 206
9.4.1.3 Peer Groups for defining Boundaries of Co-operations 208
9.4.1.4 Integrity Constraints for ensuring Consistency 208
9.4.1.5 Adaptation Policy for guaranteeing homogeneous Resources 209
9.4.2 The COBE AWARENESS FRAMEWORK 210
9.4.3 Peer Services for Construction: DocExchange 210
9.5 Application Scenarios from Structural Design 212
9.5.1 Application Scenario “Set-Up of a Networked Cooperation” 213
9.5.2 Application Scenario “Handling Dynamic Availability” 214
9.5.3 Application Scenario “Handling Adaptation Requests” 220
9.6 Conclusion 220
CHAPTER 10 CONCLUSION AND FUTURE WORK 221
10.1 Contributions 221
10.2 Limitations of the Contributions 223
10.3 Outlook on Future Work 224
REFERENCES 227
APPENDIX A: THE PI-CALCULUS 243
LEBENSLAUF 247

List of Figures

Figure 1-1: The basic structure of a service-oriented peer-to-peer architecture 1

Figure 2-1: Visualization of a peer-to-peer architecture as an overlay network across several company networks and an external ISP client 18

Figure 2-2: Principle design of SOA and Web services architectures 22

Figure 2-3: Service choreography. Each service (here: Web service) is a collaborator in a global collaboration to achieve a global goal 24

Figure 2-4: Service orchestration based on BPEL4WS. A single service (here: Web Service) encapsulates a business process. 25

Figure 2-5: Initial UML use case model motivating the use of a service-oriented peer-to-peer architecture for a supporting Groupware system 27

Figure 2-6: Exception handling in BPEL4WS 31

Figure 2-7: Comparison of Web services and components in the context of reuse. The decisive forces “leanness” and “robustness” are competitive values 36

Figure 2-8: The general structure of a component 37

Figure 2-9: The forces “robustness” and especially “leanness” influence the degree of handling potential exceptional cases 42

Figure 3-1: Meta-model of $SO_{P2P}A$ architectural style depicted as a class diagram 49

Figure 3-2: Graphical and algebraic visualization of a process in $SO_{P2P}A$ 49

Figure 3-3: Structure of ports with messages. A message is either represented by a concrete value of by a variable 51

Figure 3-4: Process communication through ports in a parallel process composition. Example depicts the composition of two components 51

Figure 3-5: The syntax of the $SO_{P2P}A$ architectural style 53

Figure 3-6: The type system of $SO_{P2P}A$ 57

Figure 3-7: Basic subtyping rules for process and basic message types 58

Figure 3-8: Meta-model of a component port consisting of three process ports..... 62

Figure 3-9: Visualization of a component in $SO_{P2P}A$ (UML component diagram). Its channel and role type constructs the port type of a port 65

Figure 3-10: Composition of two components towards a composition. The composition is augmented by façade ports (UML composite structure diagram).... 71

Figure 3-11: Information exchange between service consumer and provider during service deployment (based on UML communication diagram)..... 79

Figure 3-12: Port communication between the facades (remote interaction)..... 85

Figure 3-13: Basic execution model of a service composition..... 89

Figure 3-14: Basic execution model for composite peer services..... 89

Figure 3-15: Distributed execution model for composite services..... 89

Figure 4-1: Overview of the desired functionality for service adaptation (UML use case diagram)..... 92

Figure 5-1: The FLEXIBEAN component model (notation based on Stiemerling's dissertation)..... 116

Figure 5-2: Structure of the FREEVOLVE runtime environment..... 117

Figure 5-3: Proxy structure of FREEVOLVE is responsible for the actual management of client and server components..... 118

Figure 5-4: UML class diagram for the representation of a distributed system structure..... 119

Figure 5-5: Example for a basic constraint expression in XSemL..... 121

Figure 5-6: Action part to handle an exception (XSemL expression)..... 122

Figure 5-7: From FREEVOLVE towards multi-server environment – first sketch of a possible solution..... 125

Figure 6-1: The structural model of the DEEVOLVE infrastructure – UML class diagram..... 132

Figure 6-2: The types of a peer. A simple peer must always be connected to rendezvous peer for route out advertisements..... 133

Figure 6-3: The structure of a peer group advertisement (example of the DEEVOLVE peer group advertisement including one group peer service)..... 135

Figure 6-4: UML-based notation for modeling peers and peer groups..... 136

Figure 6-5: Model of a FLEXIBEAN instance component ("iComponent"). 137

Figure 6-6: The model of a FLEXIBEAN composition aggregating various iComponents together. The example shows a spell checker consisting of two iComponents..... 138

Figure 6-7: The model of a peer service composed out of two compositions. The example depicts a peer service for checking the spelling of a text..... 139

Figure 6-8: The structural model of a peer service in DEEVOLVE – UML class diagram..... 140

Figure 6-9: An example for a CAT-XML based composition (excerpt)..... 142

Figure 6-10: An example for a CAT-XML-based peer service (excerpt)..... 143

Figure 6-11: Model of a service composition aggregating two peer services. The example pictures a service composition yielding an extended word processor..... 144

Figure 6-12: The structural model of a peer service composition..... 145

Figure 6-13: Structure of a PeerCAT file..... 146

Figure 6-14: Example for a service composition with PeerCAT 146

Figure 6-15: Execution model for basic service composition. A third-party peer ("Triple") consumes the service composition provided by peer "Bard". 148

Figure 6-16: Execution model for a choreography composition model..... 149

Figure 6-17: Communication between peers in the choreography execution model (UML communication diagram) 149

Figure 6-18: Visualization of the open layered architecture of a DEEVOLVE peer runtime environment. Any layer can invoke operations from any layer below..... 151

Figure 6-19: The DEEVOLVE console for managing the peer environment and its corresponding peer services 152

Figure 6-20: The DEEVOLVE Mail Client 153

Figure 6-21: DEEVOLVE textual composition tool 153

Figure 7-1: Refined structural model of DEEVOLVE including concepts of adaptation policy and subscription (UML class diagram) 156

Figure 7-2: Process model for subscription (UML communication diagram) 157

Figure 7-3: Extract of a PeerCAT file for a Groupware composition. The dependency tags denotes internal dependencies between the local and used services..... 159

Figure 7-4: The DEEVOLVE PeerStore class for storing any kind of information on published and consumed peer services..... 160

Figure 7-5: Semi-formal presentation of transitions between the states of a dependency object. A state is computed by a dependency value (UML state diagrams)..... 162

Figure 7-6: The structure of an adaptation policy within a group advertisement 167

Figure 7-7: DEEVOLVE Analysis tool for visualizing consumer dependencies..... 169

Figure 7-8: The result of the dependency analysis in form of a textual wizard 170

Figure 7-9: Process of performing the analysis of consumer dependencies (UML sequence diagram)..... 171

Figure 8-1: DEEVOLVE's phase model for exception handling..... 176

Figure 8-2: Structural model of service composition including integrity constraints 178

Figure 8-3: Template for specifying an integrity constraint in PeerCAT 178

Figure 8-4: PeerCAT file representing a minimal integrity constraint for a Groupware application 180

Figure 8-5: A workflow described by an UML activity diagram. The activities are mapped to concrete peers 181

Figure 8-6: Information flow integrity defined in PeerCAT 182

Figure 8-7: Broker-based remote interaction to delegate port calls between the interface and the local part of a peer service in DEEVOLVE 184

Figure 8-8: Scenario for a broker-based approach when an exception is thrown upon remote method invocation 185

Figure 8-9: Partial structural model of DEEVOLVE encompassing broker approach . 185

Figure 8-10: Sequence diagram to visualize how an integrity is checked.....187

Figure 8-11: Structural model of a service composition with exception handlers.....188

Figure 8-12: Template for specifying exception handlers in PeerCAT.....188

Figure 8-13: UML sequence diagram to visualize the interaction between DEEVOLVE's handler service and two options 193

Figure 8-14: Sequence diagram showing the interactions among an action, the tailoring service and the composition objects.....194

Figure 9-1: Principle project constellation for a networked co-operation.....204

Figure 9-2: COM2Java bridge for mediating between FLEXIBEAN components and a COM-based COTS application (here: AutoCAD 2002).....207

Figure 9-3: Scenario of a networked co-operation based on DEEVOLVE 208

Figure 9-4: Structure of the "DocExchange" peer service 211

Figure 9-5: Structure of the DEEVOLVE architecture supporting the general application scenario of a networked co-operation.....213

Figure 9-6: Visualization of scenario 1: involvement of three engineers during the parallel planning of a supporting node. The architect (project leader) is not depicted 214

Figure 9-7: Structure of the DEEVOLVE architecture including the minimal composition during the modeling activity (shaded peers).....215

Figure 9-8: PeerCAT file of partner A defining the composition between partners A, B, D and Expert X (access ports of services are omitted) 215

Figure 9-9: Declarative description of an integrity constraint in PeerCAT for a minimal composition.....216

Figure 9-10: Declaration of an exception handler in PeerCAT 1 (actions).....217

Figure 9-11: Declaration of an exception handler in PeerCAT 2 (options) 217

Figure 9-12: Declaration of an exception handler in PeerCAT 3 (executable and selectable options) 218

Figure 9-13: Dialog for selecting options from an exception handler.....219

List of Tables

Table 4-1: Overview on the adaptation methods available in $SO_{P2P}A$ 94

Table 4-2: Auxiliary adaptation methods..... 95

Table 5-1: Overview of the concepts of the $SO_{P2P}A$ style and their counterparts in FREEVOLVE 129

Table 8-1: Overview on operations and their levels that can be applied within the definition of an integrity constraint 179

Table 8-2: Overview on actions that can be applied in PeerCAT 190

Table 8-3: Overview on the three different levels of complexity to instrument a PeerCAT-based service composition with exception handlers 195

Table 8-4: Overview on the three different levels of complexity to handle an exception in a PeerCAT-based service composition at runtime 196

Chapter 1
Introduction and Motivation

This dissertation explores the adoption of component-oriented development methods [Szyperski *et al.*, 2002] in order to master the adaptation of peer services as well as peer service compositions within service-oriented peer-to-peer architectures. These so-called *component-based adaptation* methods enable users to adapt their provided services during runtime. The same methods also serve as a way of handling *exceptional cases*, in particular the failure of a dependent service *provider* peer and, hence, a service that is part of a local service composition. A general requirement for such adaptation methods is to respect existing *dependencies* from *consumer* peers during the adaptation of a public peer service. Component-based adaptation methods as proposed in this thesis serve as an efficient way to avoid the occurrence of exceptions and thus to prevent functional misbehavior at an early stage.

In the following section, the necessity of having adaptation methods within service-oriented peer-to-peer architectures is motivated thoroughly (section 1.1). Subsequently, concrete research questions are derived in section 1.2. Based on these, the contributions (1.3) and the structure (1.4) of this work are summarized.

1.1 Description of the Problem Area

Service-oriented peer-to-peer architectures (in this work often abbreviated as $SO_{P2P}A$) refer to the class of software architectures that features a set of equal nodes, so-called *peers.* Each peer is capable of functioning both as provider and consumer of an arbitrary number of *peer services* encapsulating functions, hardware routines, or public access to documents at the same time. Consumed services can be composed to form new, more complex applications or even new services that can in turn be located and used by other third-party peers (see typical constellation in Figure 1-1).

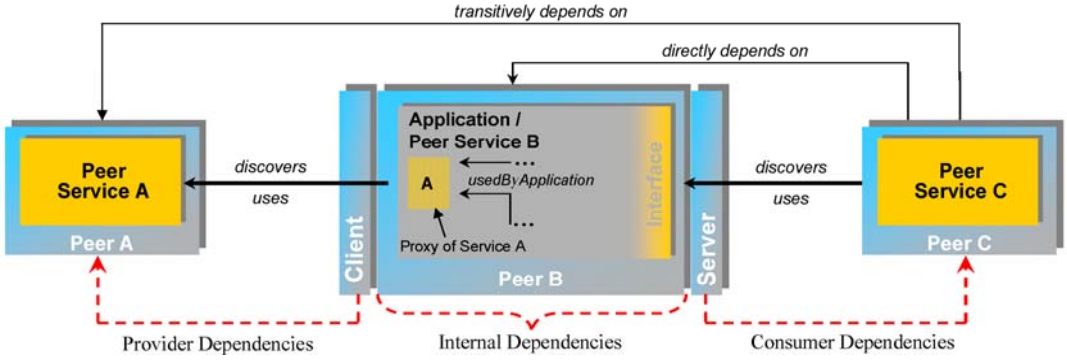


Figure 1-1: The basic structure of a service-oriented peer-to-peer architecture

This type of architecture constitutes a logical enhancement of traditional peer-to-peer architectures relying on the provision of a small number of services [Shirky, 2001], [Brookshier *et al.*, 2002] with novel concepts from the area of service-oriented architectures [Cervantes and Hall, 2005], in particular service composition. Service-oriented peer-to-peer architectures aim at supporting *collaborations* of *dispersed* working *users* who work together on a common goal. As shown in this work, distributed groupware applications can merely profit from this type of software architecture.

In fact, many conceptual and technical aspects such as non-functional requirements stemming from peer-to-peer architectures (e.g. performance, scalability, lookup, trust, and reputation) and service-oriented architectures (e.g. service discovering and description) can smoothly be adopted from the fundamental analysis of these two architectural styles. The adaptability [Henderson and Kyng, 1991] of services and service compositions has, so far, not been recognized and analyzed as a crucial non-functional requirement for these two architectural styles. This work claims that the adoption of adaptability concepts and methods is of tremendous importance for the successful deployment of service-oriented peer-to-peer architectures and is thereby absolutely worth considering. Naturally, traditional peer-to-peer or service-oriented architectures can profit from the gained adaptability insights as well.

Involving adaptability methods within a service-oriented peer-to-peer architecture is a complex problem that results from the two *viewpoints* or roles each peer may encounter during its life-cycle, namely those of provider and consumer of services. Each viewpoint poses different problems and challenges that need to be faced. Besides these local viewpoints, it is also necessary to regard a *collaboration viewpoint*. This viewpoint considers the fact that not only direct dependencies, but also *transitive* affiliations can occur between peers. In the following, consequences resulting from each viewpoint are elaborated separately. Each viewpoint exhibits a separate *problem area*.

Consumer Viewpoint: Adaptation Methods for Handling Exceptions

The handling of exceptions constitutes one of the major motivations for the utilization of adaptation methods in service-oriented peer-to-peer architectures. The importance of reacting to exceptions in these software architectures can be justified by their imposed *dynamic nature*. This dynamic nature results from the volatileness of the constituting network nodes, the so-called *peers.* Peers mainly correspond to Personal Computers (PC, notebooks) or small appliances (PDAs) whose users possess the ability to arbitrarily disconnect from the network topology without prior notice. Also, peers might become unreachable for a period of indeterminate duration due to network failures (e.g. the failure of an Internet Service Provider), disruption or loss of connectivity (e.g. the disruption of a wireless WLAN connection due to bad whether conditions), or power breakdown (e.g. the sudden failure of a Laptop's battery). Services provided by *service providing peers* may thus become unavailable as well. Consequently, affected *service consuming peers* that deploy applications relying on an unavailable service are no longer able to offer a correct run of these applications, and malfunctions may potentially occur. Any *service consuming peer* using services provided by service providing peers has to cope with the potential occurrence of exceptions.

The intermittent connectivity of peers in a given network topology is often regarded as a normal lifecycle and not as an exception [Bisignano *et al.*, 2003]. In either case, the *handling* of occurred *exceptions* is indispensable in order to avoid having long-ranging malfunctions in service applications. Exception handling is a disciplined and structured way of handling abnormal system events [Christian, 1995]. The adaptation

of a composition can be considered as such a way of handling exceptions. Existing peer-to-peer architectures utilize adaptation methods rather rudimentarily. File sharing systems such as Gnutella [Kan, 2001] or Napster [Shirky, 2001] implement automatic or so-called *adaptive* methods where the system is responsible to observe the network in order to detect exceptional cases. In such cases (here: the loss of a dependent peer during file download), an appropriate exception handler is pursued during runtime (e.g. to identify a redundant file on a different peer and continue the download process). In these systems, both the exception condition and the exception handler could have been anticipated during design of the respective peer-to-peer architecture. In many application scenarios, however, it is not trivial to determine all exceptional conditions and the respective handlers exactly during design. This is the case for service-oriented peer-to-peer architectures, where an application may be composed of many services. Even if each service provides its own exception handling mechanisms, it cannot foresee all exceptional situations and ways of handling these when assembled in a service composition. The accurate handling of an exception not rarely depends on complex *context* information, such as the location of a peer, the state of other peers, or higher-level values such as the progress of a project or individual perceptions. Trying to capture all potential exceptional cases in one single service taking into account those context values is a cumbersome activity and blows the size of a service in an unmanageable way.

Since peers are supposed to compose (new) services out of existing ones that they in turn consume from peer providers, more challenges result from possible transitive dependencies. These dependencies occur if a composed service is in turn discovered and used by a third-party peer as shown in Figure 1-1. In the exceptional case of the loss of a provider peer (peer A in Figure 1-1), any dependent third-party consumer peer (peer C in Figure 1-1) needs to be involved in the handling process. For example, affected peers could be notified to announce the exception. Again, the procedure of involving third-party peers may depend on a given context. The next problem area outlines more demands on involving consumers during the adaptation of services.

One of the main statements of this dissertation is that the implementation of existing general-purpose adaptivity models (e.g. [Oreizy *et al.*, 1999]) is not practicable for obtaining an adaptive service-oriented peer-to-peer architecture. This proposition especially proves true in collaborative scenarios, where the context that decides the way of handling an exception cannot be specified. Since human end-users are assumed to be *omnipresent* and have an active part during the runtime of a peer and a collaboration, involving end-users during the adaptation of a composition as a reaction to an exception appears reasonable. User involvement in this process must be well considered as the participant could be overstrained by the adaptation methods. If user involvement is intended, then well utilizable adaptation methods need to be conceived. These methods should meet recent standards for user adaptation (cf. [Henderson and Kyng, 1991], [Morch, 1997], [Wulf *et al.*, 2006]). So far, no approaches for holistic adaptation methods for user-oriented and autonomous adaptation are available for handling exceptions in dynamic architectures such as peer-to-peer architectures. Such methods need to be conceived for service-oriented peer-to-peer architectures.

Provider Viewpoint: Considering Dependencies during Service Adaptation

At any time, a peer provider (operator) should be able not only to adapt a composition due to occurred exceptions, but also to embrace new functionality into a peer service. For instance, user-triggered adaptation could be a reaction to changed functional re-

quirements. In addition, a user could change the appearance of an application, for instance, by modifying look-and-feel attributes or by hiding unused application parts. However, if a user carries out adaptation steps in an uncontrolled or imprudent way, potential violation of dependencies to consuming peers may arise. If an adaptation unit has not been announced or discussed with all dependent consumer peers, malfunctions may occur within the consumer's environment due to unexpected service behaviour. Moreover, service adaptation may be applied by users, who do not possess the proficiency or experience to adapt service artefacts on code-level. Consequently, adaptation methods should be designed according to well-known user adaptation principles.

A survey of the state of the art shows that no work can be found that discusses problems and requirements for the user-oriented adaptation of services within service-oriented or peer-to-peer architectures. Typical service-oriented architecture models yet provide rudimentary support for handling dependencies by having centralized service directories (e.g. UDDI) that can notify (subscribed) consumers about events whenever the used service has been changed. However, a notification only is conveyed, if the service has already been changed by service providers. Especially for strongly dependent services, such unplanned adaptations may yield massive conflicts in consumer environments. Thus, a sophisticated consumer dependency management has not been established by now. One essential implication of this work is that an efficient dependency management for service-oriented peer-to-peer architectures is indispensable for guaranteeing a stable environment where applications can operate in a reliable manner.

Collaboration Viewpoint: Establishing Collaboration between Peers

Service-oriented peer-to-peer architectures aim at supporting *collaborations of dispersed users* working towards a common goal. To achieve this goal, the involved peers (i.e. the associated runtime environments of each user) need to interact with each other in order to carry out their respective activities. According to a given working context (e.g. during the joint creation of a document), the presence of (at least a clique of) peers together with their provided peer services must be guaranteed in order to bring forward the progress of collaboration. Seen from a single peer environment, not only directly dependent peers, but also transitively affiliated peers can exist. Transitive affiliations occur, for instance, if peers collaborate according to a workflow model in which sequences of activities are defined. In order to foster the *reliability* of such collaborations, adequate rules must be defined to service as a *contract* between peers. Such a contract should regulate both benefits and obligations of the peers.

The analysis of the state of the art for this dissertation has revealed that almost no work can, to date, be found that inquiries into the act of establishing contracts for supporting collaborations of peers or between services in a service-oriented architecture. The primary challenge is to find a way of formulating such contracts and of deploying them in a service-oriented peer-to-peer architecture. Owing to the supposed dynamic behaviour of peers in that architecture, exceptions (i.e. the failure of a peer) could also violate established contracts. The handling of such contract violation, therefore, is another concern that needs to be tackled. Mechanisms for exception detection and handling must be refined in order to cover the handling of contract violation as well.

1.2 Research Questions

The presentation of the problem area in the previous section exposes various research questions that this section aims at pinpointing. The first question addresses the presen-

tation of the new architectural style of a service-oriented peer-to-peer architecture. Although it is an integration of two well-known architectural styles, the principle semantics of the elements, their correlations, and the constraints must be discussed and formulated accurately. The goal is to provide a common understanding of the underlying architecture. Question Q1 states this issue:

> **Q1**: *How can the principle semantics of a service-oriented peer-to-peer architecture be described?*

Apparently, the core questions addressed in this work concern the rationale on how to *formulate* adaptation methods and how to *integrate* these in a service-oriented peer-to-peer architecture. Two questions (Q2 and Q3) can be derived to face these concerns:

> **Q2**: *How should adaptation methods be formulated and integrated into a service-oriented peer-to-peer architecture for utilizing user-oriented adaptation of services and compositions?*

> **Q3**: *How should adaptation methods be formulated and integrated into a service-oriented peer-to-peer architecture for handling exceptional cases efficiently?*

In question Q3, the term efficiency has two meanings. First, efficiency concerns the *cost-efficient development* of applications that can be deployed in a peer environment. The goal is to exonerate the developer of peer services from utilizing bulky code for the detecting and handling of exceptions that could blow up the scale of services. Secondly, efficiency refers to *runtime-efficiency* of the resulting application. That is, the detection and handling of exceptions should not result in resource overhead.

The adaptability of peer services and service compositions necessitates the revealing of the internal structure, that is, an explicit decomposition of these artefacts.

> **Q4**: *What structure or decomposition method should be selected for modelling a peer service and a service composition?*

The next two questions tackle the debate concerning the trade-off between a purely adaptive architecture and an architecture involving end-users. This concern is relevant for the introduction of adaptation methods for exception handling:

> **Q5**: *To what extent should users be integrated during the process of exception handling in application scenarios supporting dispersed collaborations? Is a purely adaptive approach feasible?*

> **Q6**: *How should context information be extracted and considered to support the process of autonomous (adaptive) exception handling in application scenarios supporting dispersed collaborations?*

The term “dispersed collaboration” indicates a group of distributed users that work and interact with each other to achieve a common goal. This kind of application scenario constitutes the main scenario in which a service-oriented peer-to-peer architecture aims to be deployed. For this collaboration scenario, extracting context information is often not feasible due to the complexity and the distribution of relevant information. This in turn makes the process of exception handling more challenging.

The next question addresses the problem of describing and tracing dependencies between peer services. From the consumer perspective, tracing dependencies to provided

services is important for the deployment of a service composition. From the provider perspective, maintaining dependencies from consuming services is important during the adaptation of the service. Then, dependent consumer peers could be consulted or notified about a strived adaptation. Maintaining dependencies between consumed and provided services is also important for exception handling. Given, for instance, the absence of a consumed peer service that is part of a public service, potential *transitive* dependencies on other third-party consumers or on local components (e.g. a database) could be identified as well. Research question Q7 summarizes the aspects for describing dependencies between services:

> **Q7**: *How can dependencies and their relevance among service consumers, service provider, and local components be described and maintained in a service-oriented peer-to-peer architecture*?

The next question addresses the problem of analysing consumer dependencies before adapting a published peer service. The main question to be asked is:

> **Q8**: *How can a service provider respect consumer dependencies during or before a published service is adapted in a scalable way?*

While the number of provider dependencies is usually manageable, the number of consumer dependencies might necessarily become very high. The *scalability* aspect then addresses two facets: the number of consumer peers that have to be considered, and the number of peers that have to be maintained within a local peer environment. To solve this problem (question), adequate solutions need to be conceived.

The last question concerns the third problem area (viewpoint) on how *collaborations* between peers can be established as a condition (or contract) on an architectural level. In addition, it should be analyzed how the *violation of such conditions* (e.g. when a peer breaks away from an established collaboration) should be mastered.

> **Q9**: *How can a reliable collaboration between peers be defined on an architectural level and how can the violation of it be handled?*

The goal of this work is to present new contributions to the first two problem areas, that is, by giving concrete answers to the questions **Q1** to **Q8**. Enhancing reliable collaborations – expressed in the question **Q9** – is a secondary contribution that is presented reasonably. An overview of the dissertation's contributions is outlined in the next section.

1.3 Contributions

The elementary idea of this dissertation is to incorporate recent principles and assumptions originating in the research field of component-oriented methodology [Szyperski *et al.*, 2002] in the conceptual model of a service-oriented peer-to-peer architecture. This approach makes use of the general assumption that component-oriented architectures are described in terms of single self-contained building blocks (components) and *context dependencies* that describe both the interaction primitives among components and the functional dependencies on other services within a runtime environment in a *declarative* way. Given an initial set of components, this approach enables component assemblers to define compositions by means of intuitive construction methods like "define binding between two components" or "add component". At any time, dependencies are explicit and can efficiently be inspected by components, tools, or by the

runtime environment. The involvement of the component methodology in a service-oriented peer-to-peer architecture comprises four different facets:

- the structure of peer services and service composition
- the runtime environment in which services and compositions are deployed
- construction methods for external tools
- adaptation methods for the runtime environment

The proposed methods for *adapting* services and compositions correspond to the same methods as for *creating* these artifacts. In this light, methods like “delete binding between two components in a service” or “remove a component in a service” are presumed to adapt services and service compositions. The integration of these methods in the runtime environment allows for an adaptation of deployed and running services.

As a further concept, a consumer can subscribe to a provider peer as a user of one or more published services. This enables a provider peer to track its dependencies to consumers and vice versa. Thus, all internal dependencies within a peer and external dependencies to third-party peers can be traced. The availability of, say, a set of internal and external dependencies forms the foundation of a component-based adaptation environment for a single peer environment. Exceptions like the failure of peers are interpreted as the violation of dependencies on *architectural level*. In this case, the peer takes over a *reactive role* by resolving or handling occurred exceptions through the effect of component-based adaptation methods. A provider peer, of course, may take over a *proactive role* for pursuing adaptation methods on his services. The adaptation of a publicly provided peer service always precedes an analysis of consumer dependencies to ensure that no dependencies on other third-party peers are violated.

Although, from a technical point of view, both the reactive and the proactive role of a peer necessarily can be implemented as an adaptive process, this work promotes explicit user-participation at selected decision points during both adaptation processes. Component-based adaptation methods are utilized in user-oriented adaptation environments that enable users to pursue these methods to running applications.

The adoption of component-based methods for user-supported adaptability of software architectures has been studied and advantaged in the dissertation theses of Oliver Stiemerling [Stiemerling, 2000] and Markus Won [Won, 2004], both from the University of Bonn. Stiemerling elucidated the requirements for such methods in the context of component-based client-server architectures. Won’s work focused on the adoption of integrity concepts as a way for reacting to erroneous adaptation steps in the context of non-distributed architectures. The present dissertation is a continuation of these two works. The adoption of component-based adaptation methods in the context of service-oriented peer-to-peer architectures exhibits new challenges and requirements that need to be evaluated. This work features three major parts of contributions. In the following three subsections, each contribution is shortly elaborated.

1.3.1 The $SO_{P2P}A$ Architectural Style

The underlying concepts which are fundamental for creating a service-oriented peer-to-peer architecture are first formalized in terms of an *architectural style*, termed the $SO_{P2P}A$ architectural style. An architectural style serves as an abstraction for a class of coherent software architectures. It describes a set of principles for the creation of a concrete (instance of a) software architecture by making propositions about building

blocks and their relationships that need to be utilized. Also, important constraints can be defined, for instance, for defining restrictions to the topology or design elements.

This work suggests the adoption of the *pi-calculus* [Milner, 1991] for specifying the major concepts of service-oriented peer-to-peer architectures. The pi-calculus belongs to the class of process calculi that aims at formalizing a system as a set of concurrent processes. With respect to the original pi-calculus, processes are accomplished to interact with each other by sending messages over channels. The pi-calculus is an *algebra*, that is, new rules or interaction primitives for processes can be derived based on the (minimal) set of given principles. This generic approach allows for customizing the original calculus to a system with new or special requirements and conditions.

The proposed $SO_{P2P}A$ architectural style adopts the process model of the pi-calculus for modeling the underlying *component model* of that style. A single component is formalized as a process consisting of a set of *ports* (modeled as channels) that in turn can be composed to more complex composite structures. These structures form the basis for formalizing further building blocks such as peer services, service composition, peers, as well as peer groups. The interaction primitives among these integral elements are defined by reduction rules that allow formalizing the *operational semantics* for all mature types of interactions, such as internal or remote service interaction.

Reduction rules are also exercised to formalize operations for the *manipulation* of existing process structures. These manipulation operations represent the adaptation methods for adapting peer services and service compositions. The style formalizes the *reactive adapter role* (adaptation methods used as a way for handling exceptions) as well as the *proactive adapter role* (methods used for user-triggered changes).

1.3.2 The DEEVOLVE Architecture

The DEEVOLVE architecture [Alda, 2004] [Alda and Cremers, 2005] [Cremers and Alda, 2004] is a concrete instance or concretion of the $SO_{P2P}A$ architectural style. DEEVOLVE is a peer-to-peer runtime environment for providing and deploying component-based peer services within a peer-to-peer architecture. For the creation of single peer services, the FLEXIBEANS component model originally conceived by Stiemerling [Stiemerling *et al.*, 1999] for the FREEVOLVE platform has been used, which realizes type- and port-based interaction primitives for both local and remote interaction between components. DEEVOLVE extends this component model by a *service advertisement concept*. This concept allows to describe peer services by advertisements that can be published and in turn be located (or discovered) by other third-party peers.

One of the goals of the conceptual model of the DEEVOLVE architecture has been to map the rigorously formalized concepts found in the architectural style to well-established software engineering concepts. Basically, an *object-oriented structure-model* has been substantiated that embraces concepts for describing the structure of components, services, and service compositions. The identified structural objects are not only used for the purpose of clarification but are also relevant for supporting runtime concerns of services and components. They are also applied for controlling the remote instantiation process of components, guiding and delegating adaptation methods to concrete components, and for observing the remote interaction between components. An existing meta-object model for controlling component-based client-server architectures proposed by Stiemerling [Stiemerling, 2000] has served as the foundation. While some aspects of his model could be seamlessly adopted (i.e. for the remote instantiation process), additional concepts needed to be identified and integrated.

As proposed in the underlying style, the two viewpoints of provider and consumer of services can be interweaved: new peer services can be defined by the composition of local components as well as consumed peer services. For the composition of services towards more complex application and services, a special composition language called PeerCAT [Alda and Cremers, 2005] has been conceptualized. As another major achievement, DEEVOLVE also concretize the two adaptation roles described in the style in an abstract way, namely that of a reactive and that of a proactive adapter role.

In the role of a reactive adapter, a single peer environment is capable of detecting exceptions within a peer-to-peer architecture and to identify appropriate handlers that encompass adaptation methods for resolving the exception [Alda and Mitrov, 2004] [Alda and Cremers, 2004]. Exception handlers are defined on a compositional level as declarative statements in a PeerCAT description and then interpreted by DEEVOLVE during deployment. Users can be integrated in the process of exception handling at selected decision points, for instance, if handlers could not have been anticipated during design. Besides simple exceptions like the failure of peers, more complex exceptional conditions can be handled. These conditions are formulated in terms of *integrity constraints* representing *contracts* between collaborating peers. The notion of integrity constraints to describe sound service compositions at runtime has been adopted by Won's thesis [Won, 2004]. This dissertation will show how his concept needs to be refined for defining runtime contracts in a service-oriented peer-to-peer architecture.

In the role of a proactive adapter, a single peer owner is able to adapt even public peer services according to changed requirements, conditions, and so on. Before an adaptation unit can be put into effect, an analysis of existing consumer dependencies has to be carried out to suppress the violation of functional dependencies. The proposed model assumes that peers belonging to a self-organized peer group have agreed on a global *adaptation policy* in which conditions are formulated that entail procedures on how to cope with existing dependencies [Alda, 2005a]. Procedures comprise, for instance, the notification of consumers before an adaptation is carried out, or the negotiation of the scale of a planned adaptation with the respective consumers.

1.3.3 Evaluation of the DEEVOLVE Architecture in CoBE

The third major contribution evaluates the concept of DEEVOLVE on an application scenario originating from the field of construction engineering. This scenario represents a possible example for a collaboration of dispersed working users. Expertise of this engineering discipline has been gained in the CoBE project[1] [Cremers and Alda, 2004]. The goal of the CoBE project has been to study appropriate software architectures to improve the collaboration of networked co-operations within planning processes in construction [Cremers and Alda, 2004]. The DEEVOLVE architecture has been the main result of this project. The architecture serves as the foundation for further developments, such as the COBE AWARENESS FRAMEWORK for regulating distributed working activities on planning models [Alda, 2002] [Alda, 2005b] and extensions for integrating autonomous behavior using agent technology [Alda *et al.*, 2004].

This work will show how the DEEVOLVE architecture together with its utilized adaptability concepts can be used to improve the efficiency of planning processes in

[1] This project is funded in the course of the priority program 1103 „Networked-based Co-operative Planning Processes in Structural Engineering" by the German Research Foundation (Deutsche Forschungsgemeinschaft).

the special area of structural planning. The applied real world scenario used for demonstration purposes describes parts of a structural planning process for a steel bridge construction. This scenario has been established in collaboration with a project from the University of Bochum which is part of the same priority program. A first but short demonstration of the results of both projects to that scenario has been described in [Alda *et al.*, 2006]. This work elaborates in more detail how DEEVOLVE is able to support the scenario of a collaboration of dispersed working engineers.

1.4 Structure of this Dissertation

The rest of this dissertation is structured in the following way:

The goal of *section 2* is to classify service-oriented peer-to-peer architectures within the state of the art of distributed software architectures. It first summarizes the relevant architectural styles (peer-to-peer and service-oriented architectures) and discusses how they contribute to a service-oriented peer-to-peer architecture. Afterwards, the section outlines aspects of the component-based development methodology including component-based adaptation methods. The impacts of these methods for the deployment in a service-oriented peer-to-peer architecture are discussed thoroughly.

Subsequently, *section 3* uses the previously identified characteristics of a service-oriented peer-to-peer architecture as a source. The goal of this section is to present a formal architectural style, the $SO_{P2P}A$ style, which represents the core terms, concepts, constraints, as well as operational semantics for a service-oriented peer-to-peer architecture. The formalization of this architectural style is based on the formal pi-calculus.

Section 4 presents advanced concepts of $SO_{P2P}A$ for the adaptation of services, for exception handling, as well as for describing integrity constraints in a service-oriented peer-to-peer architecture. The remainder of this section summarizes related work.

Section 5 discusses the formalized concepts of the architectural style with respect to the past work regarding the FREEVOLVE architecture. It is demonstrated that some concepts of FREEVOLVE can be adopted for the conceptualization of a service-oriented peer-to-peer architecture.

Section 6 describes the fundamental concept of the DEEVOLVE architecture that serves as a concretization of the $SO_{P2P}A$ style. In this section, a structural model, the composition language PeerCAT, and the service description model are introduced. The prototypical implementation is also presented.

Section 7 illustrates consumer analysis mechanisms of DEEVOLVE debating the proactive adaptation role a single peer environment may encounter. Again, after having specified the underlying concepts, existing related work is brought together and compared with the proposed approach in this work.

Section 8 describes the exception handling mechanisms of DEEVOLVE representing the reactive adaptation role of a single peer environment. Besides a conceptual representation and an overview of the prototypical implementation, related work concerning exception handling and adaptive models for software architectures is discussed.

The goal of *section 9* is to summarize the CoBE project and to present an application scenario stemming from the area of structural design in order to demonstrate the usage of the DEEVOLVE architecture.

Section 10 finally presents the conclusion of this dissertation. The conclusion is three-folded: it embraces the conclusion of the work, sketches the limitations of the contributions, and drafts an outlook for future work.

Chapter 2
Service-Oriented Peer-to-Peer Architectures: Classification into the State of the Art

The goal of this chapter is to familiarize the reader with preliminaries and state-of-the-art concepts from the field of software engineering that are necessary for initially comprehending the notion of a service-oriented peer-to-peer architecture. Service-oriented peer-to-peer architectures exhibit elementary characteristics of service-oriented architectures (SOA) and peer-to-peer architectures (P2P). Both architectural styles, therefore, are introduced separately in sections 2.2 and 2.3, respectively. In section 2.4, a discussion is presented concerning the benefits as well as the challenges of integrating these types of architectures, yielding a service-oriented peer-to-peer architecture. This dissertation claims that this type of architecture effectively meets the requirements of distributed groupware systems supporting the collaboration of dispersed engineers. In order to justify both the rationale and the principle design elements of that architecture, section 2.4 starts with a use case model depicting the requirements of such a groupware system. During the subsequent discussion, particular attention is drawn to *handling exceptional cases* that could occur in such an architecture and to the *adaptability* of services (section 2.4.2). *Component-based* adaptation methods are stimulated as the appropriate way to tackle both requirements. General aspects of these methods together with a short survey on component orientation are outlined in section 2.5. This section discusses issues of integrating component-based principles into a service-oriented architecture. The conclusion of this section serves as an initial input for chapter 3, which will propose a formalization of a service-oriented peer-to-peer architectural style ($SO_{P2P}A$).

At first, section 2.1 will outline important preliminaries for this work. Fundamental definitions and aspects of software architectures and architectural styles are provided in section 2.1.1. These come in useful for the subsequent presentation of the different architectural styles. Issues concerning the adaptability of software (section 2.1.2) are significant to substantiate the discussion related to the adaptability of service-oriented peer-to-peer architectures.

2.1 Preliminaries

This section intends to provide fundamental definitions of terms and concepts that will frequently be used throughout this dissertation. The reader should note that none of the following definitions conform with respect to standardized or commonly agreed definitions. Apparently, some of the terms defined here (and within the forthcoming sec-

tions) can necessarily be considered as ambiguous and fuzzy. This shortcoming can be justified by the lack of a global vocabulary for research in the areas of distributed computing, software architectures, and object-oriented methodology.

2.1.1 Software Architectures

The focus of the next section is to introduce and compare distributed state-of-the-art *software architectures* from the. In fact, many definitions of the (general) term *software architecture* can be found within the relevant literature. This work regards the definition of [Bass *et al.*, 2003] as useful:

> The software architecture of a program or computing system is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them.

Substantial work on software architectures has been conducted by Mary Shaw and David Garlan from Carnegie Melon University. In an early definition [Garlan and Shaw, 1993] both authors suggest that software architectures are concerned with "… structural issues including gross organization, and global control structure; protocols for communication, synchronization, data access, assignment of functionality to design elements, physical distribution; composition of design elements, and scaling and performance". Concrete software architectures are constructed according to a set of principles that describe a family of related software architectures. Such principles are also indicated as *architectural styles*. Perry and Wolf define the style of an architecture as follows [Perry and Wolf, 1992]:

> An architectural style […] encapsulates important decisions about architectural elements and emphasizes important constraints on the elements and their relationships.

An architectural style can thus be considered as a template, while a concrete architecture complies to an instance of such a template. Formally, it specifies the *building blocks* (e.g. client, server, peer, or data base), the relationships or so-called *connectors* (e.g. a protocol, event-relationships, method calls), *constraints* (e.g. restrictions on the topology or design elements), and *rationales* of an architecture [Dustdar *et al.*, 2003].

An extensive catalogue of architectural styles is provided by Shaw and Garlan [Shaw and Garlan, 1996]. Examples of styles stemming from this catalogue are, for instance, communicating processes, event systems, pipes and filters, repositories, virtual machines, or layered architectures. The architectural styles discussed and analyzed in this work can be classified into the communication processes style. Architectures declared according to this style consist of a number of processes or objects that communicate with each other through messages. Both the client-server and the peer-to-peer architectural style are (sub-)styles of this style. These styles are elaborated and compared thoroughly in section 2.2.1.

Client-server, peer-to-peer, as well as other variants of these architectural styles tackled in this work (e.g. service-oriented architectures) impose the notion of a *service* as a constituting building block. A service is offered by a constituting process such as a server, peer, or a component that can be located and used by other processes for carrying out communication between two processes. Concrete definitions and attributes of services are usually bound to concrete technologies (such as the Web *service* defini-

tion in section 2.3.2). A more general view on services can be obtained from the areas of economics and marketing. Lovelock defines a service as follows [Lovelock and Vandermerwe, 1996] (excerpt):

> An act or performance offered by one party to another. Although the process *(i.e. the execution of a service)* may be tied to physical product, the performance is essentially intangible and does not normally result in ownership […].

Although service ownership is not prevailing, *consumer involvement*, that is, the interaction between customer and service provider, is acknowledged as a relevant attribute of service provision. Consumer involvement enables both consumer and provider to react to new consumer demands, changing processes, and so forth (see [Varki and Wong, 2003] for a more concise information on relationship marketing of services).

Beyond doubt, these marketing-related views can be adopted to the discipline of software engineering: the provision of a service is independent of the application using the service [Turner and Budgen, 2003]. However, it is the claim of this dissertation that consumer (client) relationships to service providers are not handled with sufficient attention from an architectural point of view. Especially for the *adaptation* of a public service, consumer involvement is regarded as a crucial prerequisite for meeting consumer interests (see respective research contribution in Chapter 7).

2.1.2 Adaptability of Software

This work aims at providing new insights for supporting the adaptability of distributed software architectures, in particular of service-oriented peer-to-peer architectures. The goal is to elaborate the special requirements and propose adequate methods for adapting software architecture of this architectural style. In this section, some basic assumptions concerning the adaptability of software are provided. After explaining the notion of the generic term adaptability, two more specific occurrences of adaptability – tailorability and adaptivity – are illustrated. Indeed, all three are important for the comprehension of the adaptation methods in this work.

2.1.2.1 Adaptability

An often-cited definition of the term *adaptability* has been given by Henderson and Kyng [Henderson and Kyng, 1991]:

> Adaptability denotes the capacity of a software system to facilitate modifications in response to different and changing requirements.

Henderson and Kyng stress changing requirements as the mandatory reason why software needs to be adapted. Requirements on software may change due to a couple of reasons. These could be, for instance, new emerging business processes, new standards, or new technology enablers (e.g. new programming languages). Also, requirements could be updated, if both users and developers gain a better insight after the deployment of a software product. According to typical iterative software life cycle models (e.g. Rational Unified Process [Kruchten, 2003]), changing requirements mostly result in the re-invocation of early phases of model, such as analysis, design, or implementation (see [Bruegge and Dutoit, 2004] for a good overview). Adapting software with respect to changed requirements is then mainly dedicated to sophisticated software developers. If appropriate, end-user are only involved for gaining feedback

on intermediate software products (e.g. during prototyping) or for obtaining an initial understanding of the problem domain (e.g. working observation, interviews).

2.1.2.2 Tailorability

More and more application scenarios strive for having flexible software that can be adapted to new or changing work situations during *use time* and in the context of use [Wulf *et al.*, 2006]. Here, end-users and not professionals take action to adapt individual software environments according to their personal needs. Typical examples of end-user enabled adaptation can be retrieved, for instance, in interactive system or in groupware applications [Slagter and Ter Hofte, 2002]. In either system type, graphical user interfaces are used to enhance the usability of the software. Typical operations, for instance, in a groupware office tool would be to adapt the skin of windows (e.g. the colour), to add or delete components (a new spell-checker component), or to install service packs which are mostly aimed at deploying new security functionality. End user-oriented adaptation is also indicated as *end-user tailoring*. Wulf defines the term tailorability as follows [Wulf, 1999]:

> Tailorability is defined as the possibility of changing aspects of an application's functionality during the use of an application, in a persistent way, by means of tailored artefacts (…)"

As end users cannot be expected to have proficient adaptation skills on a technical level, developers need to supply appropriate tools and techniques for providing intuitive tailoring methods. In order to satisfy this demand, research on tailoring software also orientates itself to findings from the area of software ergonomics (see e.g. [Won and Cremers, 2002]).

It is important to note that the term "end-user" often provokes the expectation of supporting totally "incline" or "ungifted" people.[2] This is certainly not true for end-user tailoring. There is a tendency to provide tailoring routines on different levels of complexity. End-users with little experience can adopt less complex routines, while better acquainted users can use more sophisticated routines. Henderson and King differentiate among three levels of user tailoring strategies: choosing between alternative anticipated behaviour, construction of new behaviour on the basis of existing pieces, and re-implementation of the system [Henderson and Kyng, 1991]. A similar differentiation is provided by Morch who puts forward customisation, integration, and extension as basic routines [Morch, 1997]. Here, customisation denotes the modification of presentation objects among a set of predefined con-figuration options. Integration refers to the creation or the combination of (existing) program behaviour that results in new functionality. Extension finally designates tailoring by adding completely new behaviour. The higher the level of complexity, the more powerful becomes the tailoring routines in order to adapt an architecture on a more granular level.

Software developers have to account for various technical aspects when realizing end-user tailoring methods in a software environment. A classical tactic allowing end-users to adapt software during use time is to *defer the binding time* [Bass *et al.*, 2003] of a software installation. According to this idea, binding decisions in an executing system not only have impact at deployment time but also at load time or runtime. Deferring the binding time towards load time or runtime enables the end user to pursue *adaptations* that affect the behaviour of a running program. Example adaptation (tai-

[2] See also the cynical remark in [Szyperski, 2002], section 24.6, p. 478

loring) techniques are based on the runtime registration of components (plug-and-play), on editing configuration files, or on component replacement (see [Bass *et al.*, 2003] for more details of each). Replacing components during runtime is the cornerstone for obtaining *component-based tailoring environments* (see section 2.5.3.3). During runtime, the structure of the binding information must be known and traceable. For a component-based environment, for instance, the structure or composition plan of a composition must be available (e.g. in a configuration file).

Deferring the binding time of a tailorable software system is at the cost of requiring extra infrastructure to support the late binding (e.g. component loader, checker etc.). In particular, late binding support during runtime requires technical environments or libraries for dynamically loading behaviour (e.g. class loading mechanisms from Java) or for manipulating behaviour (e.g. Reflection API from Java).

2.1.2.3 Adaptivity

The notion of adaptivity can be found in different publications stemming from different fields (and phases) of computer science. A first (formalized) definition by Zadeh goes back to 1963 [Zadeh, 1963] (not presented here). A more recent definition of the term adaptivity can be found in the diploma thesis of Klamar [Klamar, 2004]:

> Adaptivity is the ability of a software system to adapt itself compared to non-trivial software.

To date, many synonyms for the term adaptivity can be found, such as self-adaptation, self-configuration, self-optimization, self-healing, or self-protection (see [McCann and Huebscher, 2004] for a good overview and explanations of *some* terms). Moreover, adaptive software can be found in many fields of computer science. In the field of Human Computer Interaction (HCI), for instance, adaptivity is associated in conjunction with user modelling [Fischer, 2001]. From this field, adaptive systems can be found that possess the capability to monitor the respective user interaction [Stephanidis, 2001]. With that gained information, adaptive systems are capable of identifying circumstances that necessitate adaptation, and accordingly, of selecting and effecting an appropriate course of action. The aspired goal of those systems is to better understand and to support user navigation and learning [Marucci and Paternò, 2002].

Another area of interest for adaptive systems is the area of software architectures in general. In contrast to adaptive systems stemming from the field of HCI, the purpose of a *self-adaptable* system is to recover from any misbehaviour caused by the system. According to Oreizy and colleagues, self-adaptive software modifies its own behaviour in response to changes in its operating environment, whereas an operating environment can be anything observable from the software system such as external hardware devices, user input or sensors [Oreizy et al., 1999].

Apparently, both approaches to adaptivity illustrated above develop common ideas and intentions. Recapitulating both approaches, one can observe that typical adaptive software features some sub-system responsible for *monitoring* its internal objects (e.g. local software, users) and external objects from its environment (e.g. network, remote sensors). The system can then adapt to any (known or unexpected) incident that may occur. The adaptation often depends on the *context* of the internally or externally observed objects. With respect to the definition proposed in [Klamar, 2004], a context can be defined as follows:

> Context is any information that can be used to characterize an object or the relationship between several objects. An object may be real (physical, e.g.

> a person, or place) or an abstract object (non-physical; e.g. application, components, presentation).

Thus, depending on the context, different adaptation routines or handlers can be determined. The selection of adaptation routines is often implemented as an autonomous process incorporating, to some extent, a certain amount of artificial intelligence [McCann and Huebscher, 2004].

Adaptive software architectures merely aim at catching incidents or *exceptions* that occur within an environment and at pursuing adequate routines for resolving or *handling* them. *Exception handling* is another term used to denote this process. According to [Christian, 1995],

> exception handling is a disciplined way and structured way of handling abnormal system events.

Exception handling features allow programmers to *declare* exceptions, to treat a program unit as the *exception context* and to associate exceptions and *exception handlers* with such a context, so that when an exception is raised, execution stops and a corresponding handler is searched for among the handlers (cf. [Romanovsky, 2001] and further work of this author). There are models in which an exception can be propagated (or delegated) outside the context. Well-known approaches for exception handling can be found in modern programming languages (e.g. Java, C++). Later on in this chapter, a model for exception handling on an *architectural level* is introduced.

2.2 Peer-to-Peer Architectures

The recent popularity of peer-to-peer systems in both academic and popular scientific areas can certainly be attributed to the success of well-known file sharing systems like Napster, Gnutella, or KaZaA. Controversial discussions reflecting legal problems of these systems [Hoeren, 2002] have increased their publicity. The following sections present a state-of-the-art overview of peer-to-peer systems, covering a brief history, common definitions and characteristics, current techniques and standards, as well as a delimitation to other models for distributed computing.

2.2.1 Definition and Characteristics

A *peer* represents an equal node in a computer network, while *peer-to-peer* constitutes a model for the interaction between peers. According to this model, each peer is able to handle requests from other peers and to post requests to other peers without the involvement of a central node. A system that is made up of many peers interacting on a principle of equality is indicated as a *peer-to-peer architecture*[3].

Peer-to-peer architectures do not stand for an overly novel model for arranging distributed systems. In fact, the Internet as originally conceived in the late 1960s clearly corresponds to a peer-to-peer system. The goal of the original ARPANET was to share computing resources around the U.S. by connecting hosts as equal computing peers and not in a master/slave relationship. The Domain Name Service (DNS) and the Usenet service as fundamental services for the Internet were also designed and imple-

[3] In this thesis, the terms peer-to-peer system and peer-to-peer architecture are treated as synonyms.

mented as peer-to-peer systems (see [Minar *et al.*, 2001] for more information on peer-to-peer systems through the history of the (early) Internet).

Peer-to-Peer architectures, as understood to date, intend to link small-sized Personal Computers (PCs) and, thus, the computer resources residing on these PCs within a huge distributed network. The advance to link Personal Computers within a network can be justified by the inexorable growth of computing power of PC systems since the beginning of the 1980s. The peer-to-peer network management extension of IBM's hierarchical, mainframe-dominated Systems Network Architecture (SNA) constitutes one of the earliest attempts for the integration of computer resources of PCs. This integration was established through the common LU6.2 interface (see [Simon, 1991], [Carr, 1991] for a good summing-up of the integration). Likewise, the Grapevine Architecture by Xerox was designed to integrate distributed services across several workstations [Birrell *et al.*, 1982]. However, both the SNA extension and the Grapevine systems constituted so-called *intra-organizational systems*: only acknowledged peers (e.g. within the boundaries of a company) were able to join the network. The experience of deploying a peer-to-peer network based on SNA within a real world application scenario including almost 23,000 workstations (home and field offices) is elaborated in [Simon, 1991].

The widely-considered phenomenon of peer-to-peer systems, however, started in 1998 with the introduction of the Napster peer-to-peer system ([Shirky, 2001], [Napster, 2005]). The protocol specification of Napster (see [Scholl, 2005] for a detailed overview) is based on the popular TCP/IP protocol stack that allowed almost any PC implementing this stack to become a participant in the Napster peer-to-peer network (first releases were, at first, based on the Microsoft platform). Napster was primarily used for sharing music files (based on the MPEG3 encoding[4]) among the peers, integrating the content of nearly 1.5 million independent peers. In 2001, a first textbook was published by O'Reilly that summarized the principles, benefits, challenges, and future directives of peer-to-peer systems in an illustrative but rather non-technical way [Oram, 2001]. In this book, Clay Shirky gives a definition of peer-to-peer [Shirky, 2001] which has been cited or, to a minor degree, adopted by a plethora of other authors (e.g. [Barkai, 2002]):

> Peer-to-Peer is a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity und unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS and have significant or total autonomy of central servers.

With respect to Shirky, peers accord with *Internet*-connected Personal Computers (PC) (but also Notebooks, small appliances like PDAs) with undesignated network addresses[5]. In contrast to highly available high-end servers or mainframes, peers exhibit an unreliable character, since they can connect to or leave a peer-to-peer architecture anytime. This unreliability can be explained by the complete *autonomy* exposed by a single peer: peers are capable of deciding for which time span and to which extent they want to offer their resources to other peers [Schoder and Fischbach, 2002].

[4] See http://www.iis.fraunhofer.de/amm/techinf/layer3/index.html for more information on MPEG3.

[5] A PC may receive a different IP-address if it uses a dial-up access provider that dynamically assigns addresses for instance through a DHCP server

Peer-to-Peer architectures are often compared to and contrasted with conventional *client-server architectures*. Client-server architectures impose an *asymmetric relationship* between clients and servers: only servers are able to offer a set of services to clients, which are able use these services. Clients need to know the details of the servers that are available, but usually they are not aware of the existence of other clients. Peer-to-peer architectures are based on a symmetrical relationship between peers and, consequently, compensate for the strict distinction between client and server. The work to be done within a peer-to-peer architecture is no longer centralized, but partitioned between all peers, so that a peer consumes its own resources on behalf of others (acting as a server) while asking other peers to do the same for its own benefit (acting as a client) [Crowcroft *et al.*, 2004]. A definition of a peer-to-peer system by Sommerville also takes into account this aspect [Sommerville, 2004]:

> Peer-to-Peer systems are decentralized systems where computations may be carried out by any node on the network and, in principle at least, no distinctions are made between clients and servers.

The absence of a single server reduces the risk of having a *single-point-of-failure*, a problem that is often encountered in client-server architectures. Since the complete functionality is not concentrated on a single peer, but spread over a series of peers, peer-to-peer architectures are *more scalable* with a growing number of peers.

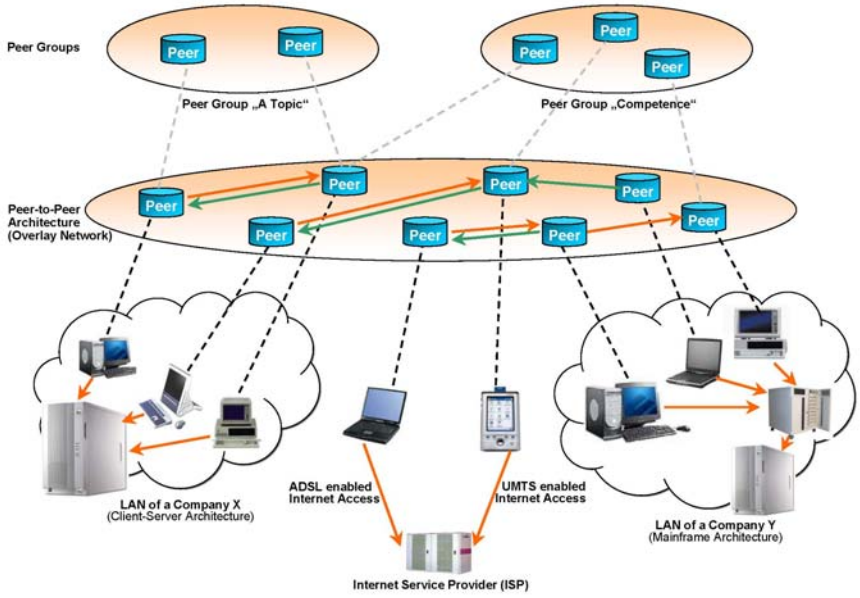


Figure 2-1: Visualization of a peer-to-peer architecture as an overlay network across several company networks and an external ISP client

These networks are often indicated as *overlay networks*. A visualization of pure peer-to-peer architecture is shown in Figure 2-1. As depicted in this figure, not only company workstations, but also external clients such as ADSL-connected Laptops or UMTS-connected appliances connected through an Internet Service Provider (ISP) can comprise a peer-to-peer system. Although peer-to-peer systems are often considered as *purely decentralized architectures* (see the last definition) without any central servers, several existing peer-to-peer architectures indeed rely on central services in their fundamental design (see section 2.2.4 for an overview and comparison of existing systems). These so-called *hybrid* or *semi-centralized peer-to-peer architectures* utilize central nodes (also denoted as super or rendezvous peer) as directories or indexes, for

instance, to maintain references to peer resources. Both hybrid or purely decentralized peer-to-peer architectures correspond to *inter-organizational architectures* as they can span a coherent network across existing network or organizational boundaries and are independent on local standards [Sommerville, 2004].

The next two sections highlight two important aspects of peer-to-peer architectures that constitute important aspects for the realization of the service-oriented peer-to-peer architectural style.

2.2.2 Self-Organizing Peer-to-Peer Architectures

The imposed independence of peer-to-peer architectures with respect to existing technical and organizational restrictions and conditions enables peers to *actively self-organize* their resulting overlay network in a collaborative manner. *Self-organization* comprehends various concepts that can be found in existing implementations of peer-to-peer architectures. In this section, two important concepts (*group-based self-organization* and *structural self-organization*) are introduced briefly.

Group-based self-organization in peer-to-peer architectures allows peers to actively organize themselves into self-governed overlays or so-called *peer groups*. This way, the boundaries of a peer-to-peer system are self-determined by the corresponding peers themselves [De Meer and Koppen, 2005]. The resulting self-governed groups are able to share, collaborate, or participate within their own private web without the assistance of a central authority [Barkai, 2002]. Peer groups can be created according to common topics, interests, competences, or any kind of computer resources. Peer groups usually prescribe an access control model in order to restrict access to any internal resource. The group (normally represented by a single or by several providers or founders) has the task to identify and to regulate access to the group. Processes for applying to, joining, or resigning from a group are also practicable [Barkai, 2002].

An evaluation of well-known peer-to-peer systems has shown that none of the investigated systems have the ability to determine their boundaries, and thus their peer group affiliations in an entirely self-determined way [De Meer and Koppen, 2005]. Surprisingly, De Meer and Koppen ignored the JXTA framework [Sun, 2005a] by Sun in their investigation. As one of the only working peer-to-peer frameworks, JXTA provides the Peer Group Membership Protocol (PGMP) for defining, publishing, as well as joining peer groups (see also section 2.2.5 for a more detailed review). Applications (and refinements) of JXTA's group concept can be found in various application domains, for instance for supporting group management in a student forum system [Halepovic and Deters, 2002] or for supporting knowledge communities (cf. [Tiwana, 2003], [Gnasa *et al.*, 2005]).

Structural self-organization is a way for optimizing the retrieval of data in a peer-to-peer architecture. In this approach, each peer accommodates a particular range of data items. A distributed index (so called Distributed Hash Table or DHT) implements a routing scheme that allows one to efficiently look up the peer where a specific data item is located. Each peer participating in a DHT obtains a small number of references to other peers and, thus, a partial view of the whole peer-to-peer architecture. This routing information enables each peer to distribute routing information towards the destination peer holding the respective data item. By mapping nodes and data into a common *structured* address space, routing to a node always leads to the data items for which a dedicated peer is responsible. A data item can always be located by routing via $O(\log N)$ hops, thus improving the efficiency of the look up operation in compari-

son to traditional flooding search methods (usually $O(\log N^2)$). A structured peer-to-peer architecture thereby is a guarantee for a highly *scalable* system.

Three implementations for realizing the principles of a DHT have received an overwhelming attention in the P2P research community: the Content Addressable Network (CAN), Chord, and Pastry. A detailed explanation of these approaches is omitted in this dissertation; the interested reader should turn to [Götz *et al.*, 2005] for a very good comparison.

Group-based self-organization will play an important part in service-oriented peer-to-peer architectures in order to regulate the adaptation of pubic peer services (see section 4.1.2 and 7.3). In contrast, scalability issues and, hence, structural self-organization are neglected within this work.

2.2.3 Reputation and Trust

The cooperative model of the peer-to-peer style may, however, break down if single peers are not provided with incentives to interact with other peers [Crowcroft *et al.*, 2004]. A peer motivated by insufficient incentives can be recognized by short uptimes as well as low quality services. The quality of a service may suffer and may be unacceptable if the service does not fulfil the quality aspects as promised to its consumers. Owing to both the (potentially) huge scale of a peer-to-peer system and the number of unknown peers within this system, a single peer is hardly able to evaluate the trustiness of all its favoured peers. In order to tackle this problem, a couple of *trust* and *reputation* models have been evolved that enable peers to distinguish highly reliable peers from poorly peers. Trust can thereby be defined as a peer's belief in another peer's capabilities, honesty, and reliability based on its *own direct* experiences. Reputation constitutes the peer's belief on the same quality values, but based on *recommendations* received from other peers (both definitions cited from [Wang and Vassileva, 2003]). Existing approaches do, in the majority of cases, utilize both models and are founded on complex mathematic models such as, for instance, the Bayesian network model [Wang and Vassileva, 2003] or the Game-theoretic model [Acquisti *et al.*, 2003]. The incorporation of such trust and reputation models into recent peer-to-peer system has, however, hardly been achieved (cf. [Sommerville, 2004]).

Reputation is an aspect for controlling and, thus, suppressing the uncontrolled adaptation of public peer services in a service-oriented peer-to-peer architecture (see 4.1.2). For the implementation of an adequate reputation system, one can fall back on aspects and techniques elucidated in this section.

2.2.4 Overview of existing Peer-to-Peer Systems

As already pointed out in the previous section, file sharing systems like Napster, Gnutella, or Freenet still constitute the most prominent peer-to-peer architectures. According to [Eberspächer and Schollmeier, 2005] (fig. 5-1), these systems belong to the *first generation* of peer-to-peer systems. Within the first generation, two separate types of systems can be identified, namely, centralized and pure peer-to-peer. Centralized systems rely on central servers to take over the role of a directory, and to store the IP addresses of the peers. Napster is a typical system belonging to this type. Pure peer-to-peer systems, on the other hand, do not rely on any central facility but utilize the technique of flooding search queries over the network (see [Eberspächer and Schollmeier, 2005], section 5.3. for a good explanation of this technique). The Gnutella sys-

tem version 0.4 and Freenet are typical representatives of this class of systems. Besides these file sharing systems, Seti@Home, a system for spreading massive computation is another prominent example of a pure peer-to-peer architecture.

It is an important drawback of pure peer-to-peer systems that they produce a huge amount of signalling traffic by flooding the request. In order to get those negative effects under control, hybrid peer-to-peer architectures aim at storing popular content together with the target address of the hosting peer at so-called super or rendezvous peers. This leads to a hierarchical topology. Hybrid peer-to-peer systems are said to belong to the second generation of peer-to-peer systems. Example systems of this class are Gnutella version 0.6, JXTA (see next section), eDonkey, FastTrack, or emule. All of those systems are file sharing systems. The Voice-over-IP system Skype[6] can also be regarded as a hybrid peer-to-peer architecture. Yet other hybrid systems have emerged for various different application domains such as, for instance, collaborative, decentralized group support (groupware system Groove[7]). The DHT-based peer-to-peer systems (section 2.2.2) also belong to the second generation.

2.2.5 JXTA – A Standard Peer-to-Peer Framework

In order to consolidate the efforts for the design of peer-to-peer architectures towards a unified architecture, Sun announced the JXTA[8] project in 2001 [Sun, 2001]. The result of this project is the JXTA protocol suite for peer-to-peer computing, which provides an open set of open protocols for developing peer-to-peer applications. The latest main release of JXTA (version 2.0) took place in 2003. Subsequently, a number of protocol revisions have been developed and published, culminating in version 2.3.5 (September 2005) [Sun, 2005a]. Developers can additionally refer to Sun's open source frameworks implementing JXTA's protocols. To date, frameworks are available for the Java platform (J2SE and J2ME) and for the language C (see www.jxta.org for more information on available downloads).

As already mentioned in section 2.2.4, JXTA features a hybrid peer-to-peer architecture. Each single peer can act autonomously as a consumer, but also as a provider of resources. Besides, JXTA spans a virtual decentral network through the deployment of a new address domain. Here, the basic addressing concept is an *endpoint*, denoting an address of a peer that implements a specific protocol of communication. A peer can have various endpoints, which enables it to communicate through multiple protocols such as HTTP or TCP. Communication between two endpoints is established through so-called *pipes*. Pipes can be compared to streams, except for the fact that they are intended as an additional layer over multiple communication protocols. The PIPE BINDING PROTOCOL is responsible for establishing a connection between two endpoints. Once the connection is established, peers can communicate via XML-based messages. Both pipes and endpoints do actually hide the complexity of each protocol from an application. The decision which endpoint is to be taken depends on the network topology (e.g. if a peer is behind a firewall, it will prefer the HTTP endpoint).

Another important concept of JXTA is the notion of *peer groups*, which provide a way for the self-organization of peers. Peers having joined a distinct group can use services, which are only available to authorized group members. The PEER MEMBER-

[6] http://www.skype.com/intl/de/

[7] http://www.groove.net

[8] JXTA is an acronym based on the word juxtapose.

SHIP PROTOCOL (PMP) regulates the access (through authentification routines) as well as the quitting of peer groups. The PEER DISCOVERY PROTOCOL (PDP) allows a peer to discover other peers, peer groups, and services in a peer-to-peer environment. All discoverable items are described in *advertisements*, which are XML-based descriptions of the respective items (comparable to WSDL for web services). Advertisements are published through the DISCOVERY PROTOCOL to all local peers residing in the same local area network (LAN) via IP multicast and to all known *rendezvous peers*. Rendezvous peers are well-known peers outside the LAN, which serve as connecting nodes to address peers outside the boundaries of a local network. More information on the JXTA platform as well as on the Java-based implementation can be found in the programmer's guide [Sun, 2005b].

JXTA is a part of the default implementation of the $SO_{P2P}A$ style (DEEVOLVE, see Chapter 6) which takes over important tasks such as service discovery and group management. This chapter will provide more information on JXTA.

2.3 Service-Oriented Architectures (SOA)

This section covers state-of-the-art aspects of service-oriented architectures. Besides a characterization of this kind of software architecture and its used standards, the main attention is drawn to the composition of services (section 2.3.3). In order to mark off service-oriented architectures from peer-to-peer architectures, section 2.3.4 outlines a discussion of these two architecture styles.

2.3.1 Definition and Characteristics

As defined by the W3C Glossary, a service-oriented architecture (SOA) is a "set of components which can be invoked and whose interface descriptions can be published and discovered" [W3C, 2004e]. The interfaces of these components satisfy the *services* available in a service-oriented architecture. Along with the actual interfaces, services are provided with descriptions, which contains a combination of syntactic, semantic, and behavioral information [Cervantes and Hall, 2005].

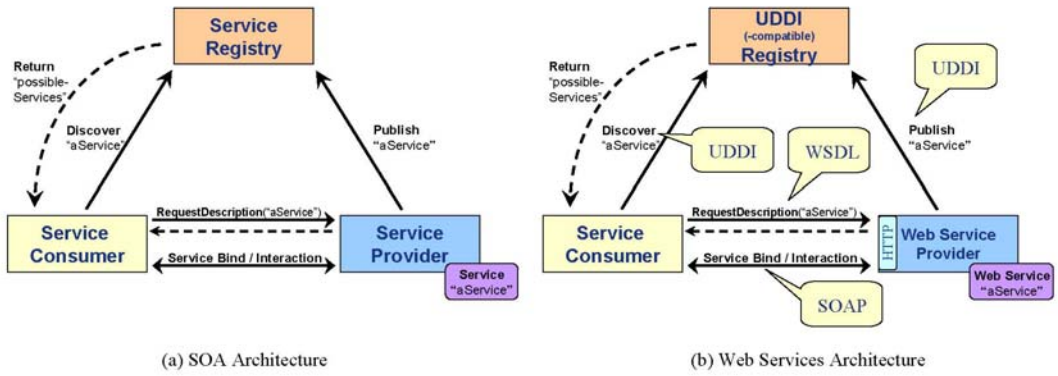


Figure 2-2: Principle design of SOA and Web services architectures

Services are either provided to end-user applications or form the basis of other services [Kaye, 2003]. Either ways, the *assembly* of services into the local environment of a *service requester* is based only on service descriptions; the actual *service providers* hosting the service are discovered and integrated into the application later, usually prior to or during application execution. Before assembling the composition, the ser-

vice requester may obtain these descriptions by querying a *service registry* to discover services based on a set of criteria that characterize the desired service (see Figure 2-2).

A plethora of technologies can actually be used to implement service-oriented architectures. Cervantes and Hall provide an extensive survey of how the technologies CORBA, JavaBeans, Jini, and OSGi can be utilized to gain such architectures [Cervantes and Hall, 2005]. However, at present on preparing, none of these technologies play a serious role on the SOA market. To date, protocols and markup languages that conform to so-called Web Services constitute the major technologies for developing service-oriented architectures. The next section treats Web Services in more detail.

2.3.2 Web Services

Today, Web services serve as the default realization of service-oriented architectures. The W3C defines a Web service as "a software system designed to support interoperable machine-to-machine interaction over a network" [W3C, 2004e]. Web Services are in particular based on SOAP [W3C, 2004b], a message format for interacting with a Web Service, WSDL [W3C, 2001], a notation for describing the interface of a Web Service, and UDDI [Oasis, 2002], a protocol for discovering and publishing Web Services. All these protocols and formats are defined on top of the universally accepted markup language XML [W3C, 2004a]. The technical realization of Web services is strongly bound in conjunction with Internet-related standards like HTTP for conveying messages through a network [W3C, 2004e]. The broad appreciation of these technologies in the industry and academic field have yielded a standardized notion for building interoperable and loosely-coupled services that are distributed within and across organizational boundaries. This work omits a detailed illustration of these protocols.

Web services assume a message-based interaction model through dedicated network endpoints or *ports*. Each port encapsulates an operation. Between ports, XML-based messages are conveyed. WSDL supports two uni-directional ways (one-way and notification) and two bi-directional ways (request-response and solicit-response) for exchanging messages between ports of different Web services. Based on a given WSDL document, developers are able to formulate the respective SOAP command for invoking a Web service.

2.3.3 Service Composition

One of the aspired goals of a service-oriented architecture is to encapsulate local proprietary legacy applications by a service. Standardized notations then allow to describe, to discover, and eventually to access that service (and, thus, the application) in a globally appreciated and standardized manner. Vendors and providers of complex enterprise applications have quickly adopted this approach as a way of minimizing the technical and (in particular) financial efforts of integrating existing applications into their business processes (*enterprise application integration* (EAI)). What is clearly missing from the core elements of a service-oriented architecture is a formalism for developing services through the explicit collaboration or composition of existing ones. Service composition accomplishes to describe typical business workflows on an abstract level. A dedicated formalism needs to come along with further attributes that guarantee typical non-functional requirements of workflows such as exception handling, transaction management, or security.

According to [Benatallah *et al.*, 2005], a service composition affords tool support consisting of at least two modules:

- A *design module* offering a user interface for specifying composite services.
- A *runtime environment* for executing a composite service and routing messages between components. Among others things, an environment should also provide for fault and exception handling as well as for dynamic service selection.

An appropriate design module requires an export format serving as a composition description that can later on be interpreted and executed by a runtime environment. Such a description typically follows a textual (declarative) representation (e.g. XML-based). The premise of a textual representation is that it can easily be interpreted and edited by design or adaptation tools. Moreover, users can necessarily read and verify a description without really knowing or understanding all technical details.

In general, one can distinguish between two ways of describing and (later on at runtime) of executing service composition, namely *orchestration* and *choreography*. Both are explained briefly in the following sections.

2.3.3.1 Choreography

A choreography (Figure 2-3) describes a collaboration between some services to achieve a common (global) goal [Benatallah *et al.*, 2005]. The control logic is distributed over the involved services. Each service then acts as a single peer. A principle activity during the development of a choreography is to establish its global goal and then to identify the interactions of the participating services in order to achieve that goal. A goal would be, for instance, the successful delivery of a car at the end of a workflow describing the buying process of a car. Choreography compositions are deployed in peer-to-peer execution environments. Here, the responsibility for coordinating the executions of a composite service is distributed among the service providers, which act in a peer-to-peer way without involving a central scheduler.

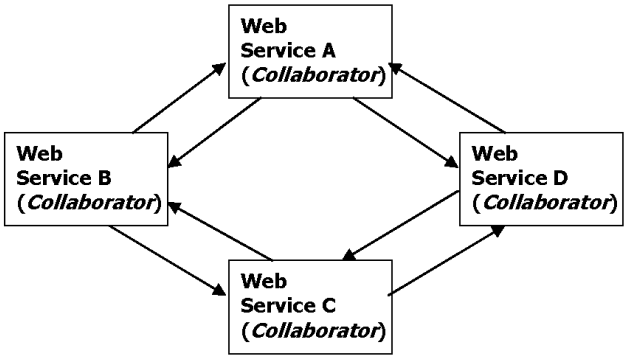


Figure 2-3: Service choreography. Each service (here: Web service) is a collaborator in a global collaboration to achieve a global goal

2.3.3.2 Orchestration

An orchestration (Figure 2-4) defines the sequence and conditions in which one service invokes other services in order to achieve its goal (w.r.t. [W3C, 2004e]). This model focuses on a single service, whereas the control logic is centralized on the respective service provider. Typical realizations of orchestration languages (see section below) feature elements for interacting with the involved services (= *activities*) in a sequence or in parallel and have options for controlling the internal flow (e.g. loop, if

condition). Hence, an orchestration describes an executable business process. Such a business process corresponds in turn to a service; that is, third-party services can publish and retrieve them. The formulation of a service as a business process consisting of elements for sequencing or branching activities, thus, reveals the internal behavior of a Web service at least to some degree. Execution engines for orchestrations are centralized and are comparable with traditional workflow engines [Benatallah *et al.*, 2005].

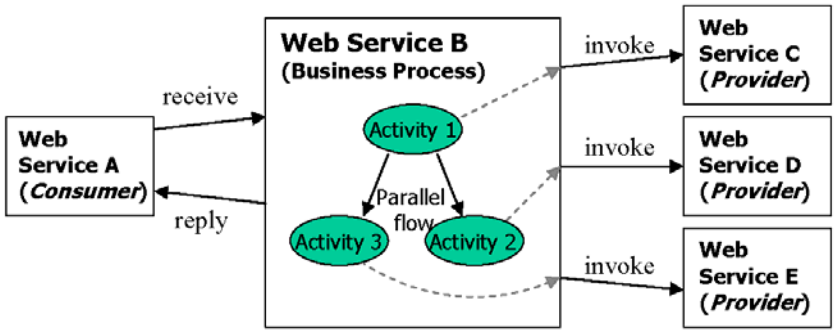


Figure 2-4: Service orchestration based on BPEL4WS. A single service (here: Web Service) encapsulates a business process.

2.3.3.3 Composition Languages for Web Services

On top of the conventional Web service "stack" (i.e. SOAP, WSDL, UDDI), new standards have emerged allowing for the composition of Web services according to the above-mentioned models orchestration and choreography, respectively. WSFL [Leymann, 2001] and XLANG [Thatte, 2001] have been early attempts for describing business processes out of various Web Services. These compositions correspond to the orchestration model. Both languages are layered on top of WSDL as they define how WSDL operations can be sequenced. Successors of these languages are BPEL4WS [BEA *et al.*, 2003] and BPML [BPMI, 2003]. Both languages feature more sophisticated concepts for business modeling than WSFL and XLANG (e.g. transaction management). Both languages only differ to a minor degree (see [Peltz, 2002] for comparison). BPEL4WS supports the definition of two types of process: abstract and executable. An abstract process is a partially ordered set of message exchanges between a service and a client of this service. It thus describes the behavioral interface of a service without revealing its internal behavior. This model corresponds to a choreography between two partners. A more powerful concept is BPEL4WS' concept of an executable process that satisfies the model of a service orchestration. It specifies the internal behavior of a service in terms of messages that it will exchange with other services together with a set of internal data manipulations. A typical scenario for such a process is as follows: whenever a message is received by a BPEL4WS executable process, the process may then invoke a series of external services to gather data before responding to the requestor (see visualization in Figure 2-4). All "invoke" activities can be structured as a sequential or parallel process or may be controlled by elements such as conditional statements. According to the idea of orchestration, the business process itself can be published as a self-contained Web service that can be used by other third-party services. This way, transitive dependencies might occur.

Further languages have emerged for solely specifying choreographies. Prominent examples are WSCI [W3C, 2002] and WS-CDL [W3C, 2004c]. WS-CDL aims at increasing both the interoperability with and the reuse of existing compositions. To do so, WS-CDL is not solely based on WSDL as an interface description. Moreover, WS-

CDL features a clear semantics as most constructs can be expressed by pi-calculus expressions. The pi-calculus also plays an important role for formalizing an architectural style describing a service-oriented peer-to-peer architecture (chapters 3 and 4).

Today's established runtime environments for Web Services support well the widely-established Web Services stack (SOAP, UDDI, WSDL), but only few support service composition languages [Benatallah *et al.*, 2005]. Most runtime environments like the IBM WebSphere product family realize centralized server-sided rather than peer-to-peer environments. In fact, most environments found today are research prototypes or open source projects (e.g. ActiveBPEL[9]). For a more detailed overview of such environments see [Benatallah *et al.*, 2005] and [Hantschel *et al.*, 2006].

2.3.4 Comparison: SOA vs. Peer-to-Peer

At first glance, the peer-to-peer and service-oriented architectural style exhibit commonalities. Both styles entail the concept of a service that encapsulates local resources that can be published, looked up, and eventually accessed from elsewhere. Peer-to-peer architectures, however, focus on the integration of simple resources via protocols aimed at providing specific *vertically* integrated functionality [Foster and Iamnitchi, 2003]. Thus, peer-to-peer architectures get by with a small number of services. A file sharing architecture for instance like Gnutella basically incorporates a single query service for discovering data in a peer-to-peer network[10]. In contrast, service-oriented architectures potentiate the provision of a broad range of services. Accordingly, one of the huge challenges of peer-to-peer architectures to retrieve data that matches a concrete peer's query is shifted to the problem of retrieving a suitable description of a service that matches a client's query. In order to optimize the retrieval of both (data or services) descriptions numerous new ways are proposed. These range from specifying conditions and procedures for matching services [Zaremski and Wing, 1997], adding meta-data to a description [Dornfest and Brickley, 2001], towards enriching service descriptions by taxonomy descriptions (ontologies) [Radetzki and Cremers, 2004].

In contrast to Web Services (or to service-oriented architectures in general), peer-to-peer architectures provide a much looser classification of who is a provider, a consumer, or a registry. A host within Web Service-based architectures may necessarily represent a producer and a consumer, but not a registry [Schneider, 2001]. In most web service architectures, however, the delimitation between a provider and consumer is even abolished. From there, many authors state that Web Service architectures meet the conventional client-server architecture (see discussions in [Barkai, 2002]; [Wojciechowski and Weinhardt, 2002]; [Bussler, 2003]). In fact, the SOAP specification in conjunction with WSDL allows to define the external interface of a server, but not of its clients, violating the general peer-to-peer approach of having equally interacting nodes [Bussler *et al.*, 2003].

Web Service implementations usually deploy a single global registry that a client can request for querying service entries. In peer-to-peer architectures, querying for service descriptions (*advertisements*) is *self-organized* by the contributing peers. For doing so, a query to an advertisement is (with respect to a given distributed data structure modeling the peer topology) repeatedly disseminated to neighboring peers until the respective advertisement has been located (cf. section 2.2.2). Given a huge scale of

[9] http://www.activebpel.org/

[10] Auxiliary services for instance for sending alive signals, routing messages, and so forth are not regarded.

peer-to-peer architectures, efficient routing and data retrieval algorithms are indispensable. These concerns are normally out of the scope for Web Services architectures.

Other authors such as Yau express the grade of availability of services as one crucial factor for distinguishing peer-to-peer from Web Service architectures: while peers can drop off any time and, thus, exhibit no high availability, Web Services are assumed to reside on highly available web servers. Availability, reliance, and reliability are important issues for Web Services, as consumers may in turn take these services to create their own reliable business processes [Yau, 2001].

According to David Barkai, the additional value of peer-to-peer compared to web service architectures can be seen as the ability to organize working groups in a virtual sub-network, facilitating the direct collaboration between people [Barkai, 2002]. There are in fact no known endeavors for the definition of a protocol for the grouping of web services similar to the Peer Membership Protocol, which is present in the JXTA protocol suite for peer-to-peer applications (see section 2.2.5).

2.4 Discussion: Service-Oriented Peer-to-Peer Architectures

This section discusses possible chances, consequences, and technical issues of *service-oriented peer-to-peer architectures*. This type of architecture is proposed to serve as an effective software architecture for groupware systems supporting distributed organizations. The requirements of groupware systems are analyzed and depicted in section 2.4.1. Based on these requirements, general properties of a service-oriented peer-to-peer architecture are derived and presented in section 2.4.3. Following this introduction, the most relevant concerns of that new architectural style (service composition, exception handling, as well as adaptability) are highlighted in separate sections.

2.4.1 Starting Point: Requirements of a Distributed Groupware System

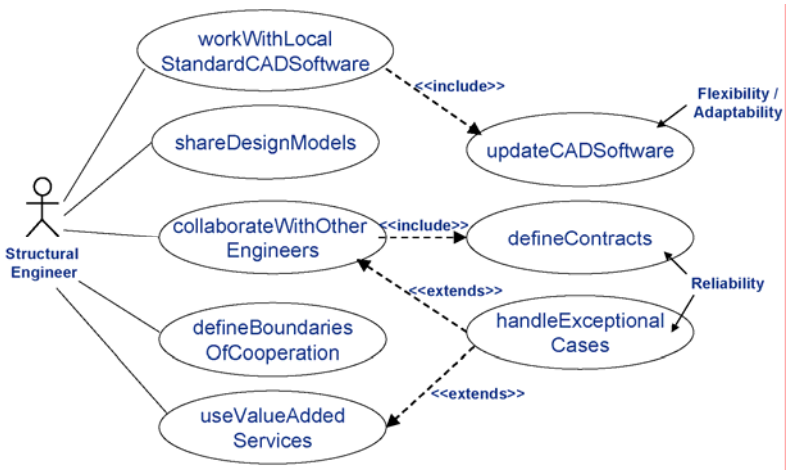


Figure 2-5: Initial UML use case model motivating the use of a service-oriented peer-to-peer architecture for a supporting Groupware system

The idea of a service-oriented peer-to-peer architecture has emerged during the analysis of requirements in the CoBE project (see chapter 9 for more details). The goal of this project has been to elicit requirements for a distributed groupware system that aims at supporting the collaboration of dispersed engineers in complex projects in construction engineering. Figure 2-5 depicts the resulting use case of this analysis

process showing both functional and non-functional requirements of such a groupware system. More information on the requirements of such a system are provided in section 9. This section only outlines a condensed presentation.

The analysis process has revealed scenarios, in which single *end-users* (here: structural engineers) aim at providing as well as consuming any kind of resources (here: design models that are available in an XML-based way) with other users. The exchange of models is termed as a sharing of design models (use case "sharingDesignModels"). Structural engineers prefer working with their local standard software (CAD software, such as AutoCAD), in which the respective design models are produced and viewed. Owing to the virtual project constellation, no central infrastructure, for instance, for storing design models in a central repository is available. So, all design models stay within the local groupware environment. Besides sharing documents, all engineers should be capable of providing and using *value-added services*, for instance for checking the correctness of design models. Also, typical groupware services such as mail, printer service, etc. could be provided by these services. These services should potentially be composed of the local applications or services (e.g. access services to the AutoCAD installation) as well as of other services consumed by third-party providers. Engineers should be able to add services and to define compositions on their own by means of appropriate tools.

Owing to the collaborative character of such scenarios (users interact with each other to reach a common goal) it is necessary to define permanent dependencies between participating users. Dependencies can be interpreted as dependencies between services that allow users to continually share resources, using value-added services and so on. On top of that, the boundaries of a distributed planning constellation should be well-defined, in particular to restrict the unauthorized access to any resource.

In order to realize these core use cases, three additional supplier use cases are suggested in the use case model of Figure 2-5. These use cases are important to implement relevant *non-functional* requirements, that is, additional quality properties of the integrated architecture. In order to have a flexible groupware application, users should always be able to adapt (or to update) their local installation according to new demands (e.g. upon release of a new AutoCAD version). Adaptation should be mastered by the engineers themselves, that is, without any intervention from a developer.

For supporting the reliability of collaborations, it is beneficial to define contracts between participating engineers. Contracts entail both benefits and obligations for all involved engineers. Since engineers are expected to run their groupware installations on conventional personal computers or laptops, these contracts should merely be based on the availability of resources. The availability of resources needs to be defined according to distinct working contexts, in which the collaboration carries out important planning activities (e.g. the parallel construction of a steel bridge deck).

During the use of a local groupware application, misbehaviour can occur when dependent services becomes unavailable. Such exceptional cases must be handled circumstantially at runtime, in particular if a defined and valid contract is violated. Since the handling process depends on the working context, a user needs to be involved at this stage in order to assess the impact of the current exception on the given context. Based on this assessment, users should be capable of selecting a distinct handler for handling the occurred exception.

2.4.2 Discussion: Appropriateness of Peer-to-Peer and SOA

At first sight, a peer-to-peer software architecture seems to be suitable as a fundamental software architecture for a distributed groupware system in order to meet the requirements of section 2.4.1. A local groupware installation could be regarded as a peer that is capable of providing as well as consuming resources and services. A peer in this case is a conventional personal computer that is directly associated with a human user. Each peer maintains his resources locally, that is, there is no central element for sourcing out resources. Another typical characteristic of peer-to-peer architecture is the ability of peers to self-organize the resulting architecture for determining the boundaries of (sub-)collaborations. Looking forward to a concrete implementation, frameworks such as JXTA could necessarily be adopted for a concretization.

Peer-to-Peer architectures, on the other hand, exhibit restrictions concerning the implementation of service composition aspects. In addition, issues concerning exception handling and adaptation of services can hardly be found in recent studies on peer-to-peer architectures. In contrast, service composition and exception handling are two major aspects of service-oriented architectures. In the next three sections, these three design aspects (composition, exception handling, and adaptability) are discussed extensively. The goal is to find out reasons why peer-to-peer architectures fail to meet the special requirements of the aspired groupware system. Besides, it is verified to what extent peer-to-peer architectures may profit from existing approaches by service-oriented architectures to implement these aspects. Regarding adaptability of services, the thesis will show that both architectural styles do not cover this non-functional requirement in a satisfying manner.

2.4.2.1 Aspect: Service Composition

A crucial requirement of the aspired groupware system is to enable *peer operators* to assemble diverse (value-added) services together with local application blocks towards new and more complex applications. Such a composition is also suitable for defining temporary collaborations between peers. One essential statement of this thesis is that both general studies and concrete implementation of peer-to-peer architectures (JXTA, proprietary solutions) do not address service composition at all. For the time being, the lack of composition notations and runtime environments can be justified by the prevailing application scenarios available for peer-to-peer architectures. It turns out that services provided by peers are usually consumed directly by end-users rather than other services. The number of services is usually rather small (mostly one business service, e.g. a service for sharing documents), so that mostly a composition is not applicable. Another reason is certainly the assumed dynamic nature of peer-to-peer architectures. Given the risk of losing the connection to a remote peer service, reliable service compositions might hardly be put into practice, at first glance. As already motivated in the introduction, adequate exception handling mechanisms are needed for the execution of service composition in a peer-to-peer environment (see next section).

Apparently, existing notations and runtime environments proposed for Web service composition (section 2.3.3) could be contemplated for the composition of services provided by peers. Approaches following the *orchestration* model (e.g. BPEL4WS) are appropriate for construction applications consisting of many groupware services (e.g. for sharing design models) and value-added services (e.g. a consistency checker) within a local groupware environment. Orchestration-based composition, however, suffers from having no collaborative support as the focus is set on one service only.

Unlike in choreography based approaches (e.g. WS-CDL), no global goal can be defined that all services (and, thus, all peers) have to achieve. Working towards a common goal is a crucial requirement for a group of engineers in the above-illustrated use case. So, a suitable runtime environment for the illustrated groupware application not only has to provide integrated mechanisms allowing users to define orchestrations, but it also needs to involve its peer environment in an (existing) collaboration that works towards a common goal.

Although a collaboration between engineers is usually able to define a common goal (e.g. finishing the design of a steel construction), its interactions cannot completely be anticipated and derived from that goal. Interaction between services depends on many contexts, including spontaneous user decision, user intuition, external incidents (e.g. rules for constructing models have changed), or the absence of existing partners. Unlike the design approach of a choreography, no exact workflow can be determined between all involved services. What is rather needed is a *structural composition* between services where dependencies are defined, but *no* sequence of execution. The sequence of service activities is initiated by the involved users.

An important aspect of supporting the reliability of collaborations is to establish contracts that define benefits and obligations in terms of the availability of peers. In the context of SOA, such contracts are often indicated as service level agreements (SLA). At the time of preparing this thesis, no piece of research could be identified on this topic, nor is there any standard template for describing such agreements. Moreover, no common opinions are available on how to analyze SLAs in order to see whether they are satisfied or violated. Typically, SLAs are negotiated and arranged in terms of external contracts (e.g. oral or written). State-of-the-Art languages, however, do not allow the declaration of contracts on the basis of service compositions. In the context of the illustrated use case, contracts are essential to define valid compositions according to a given working context.

Although the operations belonging to a Web Service can be formulated exactly (by means of WSDL), and even though orchestration-based composition languages accomplish to define how these operations can be ordered (including iterations, concurrency, iteration statements), Web Services are comparable to “virtual components”. Here, their inner implementation is hidden (encapsulated). Especially in a choreography, *internal dependencies* from a public service to local building blocks (e.g. database connections, graphical user interfaces, persistent storage and so on) or other services cannot be extracted. A fine-grained decomposition of all building blocks as (locally available) Web Services is certainly too tedious and complex, since for each service bindings to a concrete programming language need to be defined. Since externally consumed services can be used to set up other public (Web) services, even transitive dependencies could occur (see Figure 2-4). To maintain these dependencies is of major importance, especially in dynamic peer-to-peer architectures with a high degree of fluctuating service providing peers. In case of a service failure, dependencies on both affected local building blocks and other services can be clearly identified.

This section arrives at the conclusion that existing approaches to service composition are valuable for the adoption in peer-to-peer architectures. However, important aspects that are necessary to satisfy the requirements for both the characteristics of a peer-to-peer architecture and the imposed conditions of the use case are missing. Besides, it has been pointed out previously, that an exception handling to detect unavailable peers becomes essential. The next section analyzes how the service composition approaches the handling of unexceptional cases.

2.4.2.2 Aspect: Exception Handling

The handling of exceptions is a crucial requirement for the development of service-oriented architectures when used for realizing real business processes. The architecture must take into consideration how the system will react if there is an error or if the service invoked does not respond or is unavailable. According to Peltz, almost 80% of the time spent in building business processes is spent in exception management [Peltz, 2002], thus pinpointing the relevance for having exception-aware architectures. The relevance of exception handling is even increased for peer-to-peer architectures. While typical Web service-based architectures yet assume serious Quality of Services (QoS) aspects concerning the availability of services, peer-to-peer architectures have to cope with unreliable and temporarily connected service providers.

Existing peer-to-peer architectures like Gnutella only have basic exception handling models. The only way of handling the exception of a lost provider peer is to locate an alternate provider peer that holds a redundant data document and then to proceed the download process. Owing to the lack of service composition models for peer-to-peer architectures, models for handling exceptions within service compositions are missing.

Exception handling for service-oriented architectures (i.e. Web services) is realized by all well-known service composition languages. These languages accomplish to define handling mechanisms for exceptions *in a declarative manner* within the context of a service compositional description. Exception handling is used to capture unexpected behaviour of a service. Exceptions may be invoked by incoming messages that correspond to request/response or one-way operation in WSDL or through pre-defined alarms that go off after user-set times (e.g. onAlarm or onTimeOut). Exceptions within services can only be caught and handled if they have been associated as part of an *exception context* that either corresponds to the whole process (cf. WSCI) or to a part or scope of it (cf. BPEL4WS). An exception handler that belongs to a context comprises the *exception condition* (message or alarms) as well as the actual *handler* that defines an activity set to perform should that exception occur. Figure 2-6 demonstrates the use of an event handler (BPEL4WS) to enhance the termination of a process through an external message.

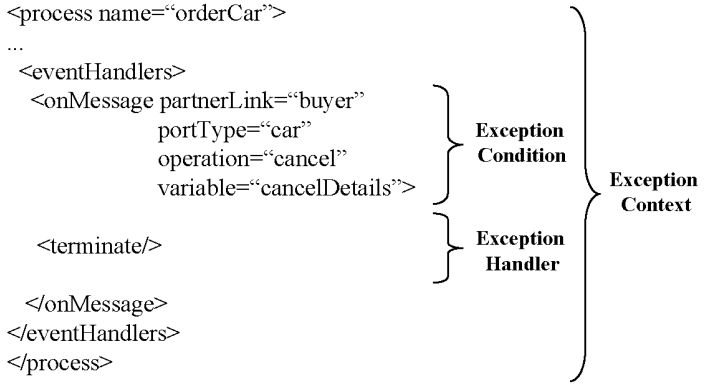


Figure 2-6: Exception handling in BPEL4WS

Obviously, a combination of message-based and alarm-based exception handling could be used to utilize a monitoring mechanism to determine if a service is available and to pursue a respective handler if not. Languages such as WSCI also allow for locating alternate services (locate operation) in a registry such as UDDI and for late-binding a selected one into a given composition (see [W3C, 2002]).

A general drawback of the exception handling concepts of all inspected composition languages is that for a given exception condition only one exception handler can be associated. Given the definition of various handler statements with respect to a single condition (e.g. a second onMessage tag in Figure 2-6), the semantics of these languages entail that the first handler is to be invoked. All others are ignored[11]. So, both the exception condition as well as the exception handler have to be exactly *anticipated* beforehand. While this is possible for establishing the exception condition, foreseeing the exact procedure how to handle an exception is a non-trivial task. The decision which handler is preferable also depends on a plethora of different *exception context information.* Prominent examples for context information could be the time, place, status information of external peers or services, and so forth. As mentioned in the introduction section of this dissertation, identifying this information in advance is somewhat difficult for dynamic project settings. The introduced use case also assumes working contexts that cannot be described or identified entirely. Rather, a working context depends on human perception or on complex information such as the progress of a project, the availability of partners, on external facets, and so forth. Putting such information into a context is certainly very hard if not impossible.

A further disadvantage of service composition languages presented here is the absence of mechanisms to *involve users* at selected decision points. Users could, in particular, be involved during exception handling. In this light, a list of multiple exception handlers belonging to a single exception condition could be defined, from which a user could select the most suitable one during runtime after the exception has been caught. By doing so, the service assembler is not forced to fully anticipate the most appropriate exception handler during composition time. Instead, multiple variants can be defined and the final decision is up to the user during runtime. He selects the most suitable handler according to given (perceived) context.

From a technical perspective, user involvement is certainly practicable by modelling methods that encompass *proprietary* bindings to an implemented user dialog. Such a dialog could then be used to realize some user-to-system interaction for resolving the exception. However, there are no commonly agreed and *standardized* methods to model explicit user involvement within the process of exception handling. Having a standardized way for modeling user aspects would increase the *portability* of service compositions to other host systems (e.g. embedded systems). Depending on the capabilities of the host system, appropriate tools could be implemented for representing the necessary elements for involving users.

A further requirement results from the potential transitive dependencies within an orchestration composition (Figure 2-4). Here, not only the local composition but also third-party peers could be affected by an exception. This effect occurs if there are internal functional dependencies between services consumed and services provided. Consequently, exception handling is no longer a local process but becomes a global process in which many peers have to be taken into account. However, no languages inspected for this work incorporate mechanisms to notify dependent services (or peers) about the occurrence of an exception. In fact, users should also be notified about an occurred exception so that they are able to potentially react to an exception themselves, as well. This is again important due to the *omnipresence* of the user during the execution of a service and, thus, during the collaboration with other users. A conse-

[11] Accurate elucidations on these restrictions can be found in the respective specifications for BPEL4WS ([Bea et al., 2003], section 13.5.1), BPML ([Arkin, 2002], section 9.2, p. 47) and WSCI ([W3C, 2002], section 3.9.1)

quent requirement for a runtime environment is to allow *consumers* (a human being or a technical system) of services to subscribe to a list so that for any exception, dependent peers can be identified and notified instantly.

This work suggests the adoption of basic ideas of exception handling from established languages for service composition in SOA. However, aspects concerning user involvement during exception handling are of particular importance.

2.4.2.3 Aspect: Adaptability

A major observation that occurred during research of this thesis is that supportability (including adaptability) aspects are hardly regarded in both peer-to-peer and service-oriented architectures. A similar observation is stated by [Poizat *et al.*, 2004]. Considering other non-functional requirements, most research work carried out to date for peer-to-peer architectures is concerned with *scalability* issues (e.g. CAN, see section 2.2.2) as well as *security* issues (i.e. trust and reputation models in peer-to-peer architectures in section 2.2.3).

This work aims at providing novel contributions concerning the adaptability of service-oriented architectures. In particular, the adaptability of single services should be analyzed. The adaptation of a single service can, for instance, be undertaken by adding, changing, or removing ports from the interface, by changing the internal implementation, or by entirely removing the service. Either way, the adaptation of a public service might affect the functionality of other compositions that rely on the *old* version of it. To avoid any misbehaviour, consumers should be enabled to subscribe to the provider site or to a directory, in order to at least become notified about any changes. Such a subscription mechanism is defined in the UDDI specification v3.0.2 [Oasis, 2002][12]. Subscription provides consumers, known as subscribers, with the ability to register their interest in receiving information concerning changes made in a UDDI registry. These changes can be scoped based on preferences provided with the request. Subscription scenarios comprise use cases, for instance, for notifying consumers whenever a service becomes available that conforms to a set of criteria. Also, consumers can be updated on a particular service whenever it is altered in any manner, including deletion. UDDI provides APIs for both consumers (registration and polling for new events) and providers (pushing changes on services to the directory).

The adoption of a central service directory in a service-oriented architecture counteract the idea of having a decentral architecture without any central authorities. For this reason, consumers should be enabled and also convinced to subscribe directly to the provider peers as indicated in section 2.4.2.2. A local *subscriber store* could therefore be used for both exception handling as well as adaptation management.

The actual subscription and notification mechanism of UDDI exhibits further restriction. The notification of consumers is pursued right after a service *has already been* altered or deleted by the provider. Hence, consumers can be notified, but are not given time to react to effected changes accordingly. An improvement would be to *announce* the date of an adaptation, so that all consumers can prepare for the forthcoming changes. Such an announcement mechanism could also support consumers to *negotiate* details or rationales of the planned adaptation or to send an explicit agreement or disagreement statement to the provider. Since concrete implementations of service-oriented peer-to-peer architectures intend to support the collaboration of dispersed

[12] see section 5.5 and appendix C of the UDDI specification for more information on subscription management.

working people, the conceptualization of such a consumer-oriented adaptation model would definitely be beneficial.

The adaptability of service-oriented peer-to-peer architectures also requires the inspection of usability requirements in order to facilitate the adaptation of services for system operators. Owing to the potential dependencies on third-party consumer services within an orchestration (see Figure 2-4), adaptation now becomes a complex and critical task: uncontrolled adaptation of a service could lead to misbehaviour if dependencies are not evaluated carefully. Uncontrolled adaptation would also decrease the trustworthiness and the reputation of a service provider within a collaborative peer-to-peer architecture. To avoid these consequences, sophisticated end-user adaptation or tailoring mechanisms (see section 2.1.2.2) need to be conceived. To date, tailorability concepts for service-oriented architectures have not been considered. For a new architectural style meeting the above-mentioned requirements, thus, completely new thoughts on tailoring mechanisms need to be considered. Resulting tailoring mechanisms could also be adopted for regular service-oriented architectures.

2.4.3 Suggestion: Service-oriented Peer-to-Peer Architecture

This dissertation suggests the adoption of a *service-oriented peer-to-peer architecture* for fulfilling the demands of a distributed software system such as the groupware system explained in section 2.4.1. This type of architecture incorporates the essentials of both peer-to-peer and service-oriented architectures into one integrated architecture. The corresponding (*novel*) architectural style is a reasonable continuation of the peer-to-peer architectural style as discussed so far. One particular statement of this thesis is that future peer-to-peer application scenarios will not only include a single service (e.g. data retrieval). More and more, scenarios will also demand for a *class of services* such as secure document exchange, a shared calendar, or shared whiteboards (see also [Steinmetz and Wehrle, 2006] for an overview of potential future applications of P2P). This future direction has already been motivated by the recent *de-facto* standard for peer-to-peer development, JXTA. This work therefore is a continuation of the work done for the JXTA framework.

Due to the variety of services, the development of services through composition of existing ones will become an important issue. While service composition in a service-oriented architecture rather describes (automatic) business processes, service composition in a service-oriented peer-to-peer architecture describes *collaborations of dispersed working actors* working towards a common goal. In contrast to SOA, composition refers to *structural* composition of services. Here, only static bindings between services, but no information concerning the flow of control are defined.

In this new software architecture, users are *omnipresent*. This has influence on many activities that are necessary to set up and, at runtime, maintain a service-oriented peer-to-peer architecture. Another important constraint is the *dynamic* behaviour of single peers, as peers normally represent conventional personal computers. The following definition of a service-oriented peer-to-peer architecture takes all aspects mentioned above into consideration:

> "A *service-oriented peer-to-peer architecture* features a set of *self-organizing peers* serving as a runtime environment for providing, consuming, and composing *peer services*. *Omnipresent users* run peers that are characterized as fluctuating networks nodes. Owing to the direct associa-

> tion between users and peers and due to the expected dynamic behaviour of *both*, users are actively involved to set up and, during runtime, to *maintain* the resulting architecture."

Activities for setting up an architecture refer to composing and deploying services. The composition of services also includes the process of establishing contracts between peers regulating the availability of peer services. During runtime, adapting services and service composition as well as handling any exceptional cases resulting from dependent services constitute the main activities for maintaining an architecture.

2.4.4 Concluding Remarks

The requirement of a service-oriented architecture to involve human users during both the adaptation of a service during runtime and the handling of an occurred exception implies the provision of tailoring routines suitable for that architecture type. For realizing both activities, the structure of both the composition and the service must be explicitly declared and traceable (see section 2.1.2.2). While the structure of a composition can be easily referred (declarative description), the structure of a service is not provided by default. SOA implementations like Web Service architectures make assumptions about the structure of an interface (i.e. WSDL) and the interplay of different services (i.e. composition languages like BPEL4WS), but make few assumptions about the internal implementation of a service (*virtual components*). Consequently, for realizing tailoring mechanisms for a service-oriented peer-to-peer architecture, a structural model for a service needs to be conceived. Based on this structure, adequate tailoring routines can be derived and proposed. By default, the original definition of a service-oriented architecture makes no assumptions on how to decompose a service and how to arrange tailoring routines.

The message of this work is that *component-based peer services* together with *component-based tailoring methods* [Stiemerling *et al.*, 1999] [Won and Cremers, 2002] serve as the most effective way to create peer services and to adapt them during runtime, respectively. Tailoring methods could not only be used for adapting services, but also for adapting service compositions and for restructuring compositions after the occurrence of an exception. This way, users who have skills in tailoring component-based applications could easily learn and adopt methods for handling exceptions.

Before carrying on with a more detailed explanation of interweaving the service-oriented and the component-oriented model, some more aspects on component orientation are provided in the following section.

2.5 Component Orientation

The rationale behind component orientation is to build software systems composed of single ready-made software entities called *components*. This vision originates from the field of engineering science, where one has always been committed to design systems from existing components (for instance, computer or hi-fi systems). The next sections will give a brief outline of the state of the art of component-orientation in software engineering.

2.5.1 Definition and Characteristics

The idea and the benefits of software components have first been elucidated by McIlroy in 1968. In this work, components are considered to provide "routines to be widely applicable to different machines and users" [McIlroy, 1968]. Although the merits of software components have certainly been well-founded, it took over two decades to establish this technology as a widely-accepted approach to build software architectures in a flexible and cost-effective way. In more recent publications, components have been assigned various but mixed characteristics. This can be seen, for instance, in the discussion concerning the structure of components: while some authors, such as Jacobsen, regard components as a binary building block [Jacobsen *et al.*, 1992], others allow components to be available as source code [Sametinger, 1997].

A widely-accepted and commonly used definition of a software component has been stated by Szyperski: "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [Szyperski *et al.*, 2002]. With respect to his work, software components also possess the following additional properties:

- Components can be composed and deployed to form concrete compositions or applications. *Deployment of components* comprehends the addition, the replacement, but also the deletion of components. Resulting component-based compositions thus do not correspond to monolithic applications as imposed for instance by the object-oriented paradigm.
- Components exhibit *neither an identity nor a persistent state* that facilitate an increase in both the degree of reuse and the maintenance of a composition.

Apparently, Szyperski's model of a software component (including the mentioned properties) has been adopted by recent component technologies only to some extent. As such, some component technologies rely on having components that maintain a persistent and user-bound state (cf. "Stateful SessionBeans" or "EntityBeans" from the Enterprise JavaBeans (EJB) model [Sun, 2002]).

Components are often compared with other software building blocks from the field of software engineering, such as objects, classes, modules, or types. A good distinction between the notion of a component and these building blocks as mentioned above can be found in [Frank, 1999] and [Sommerville, 2004] (component vs. object). A major difference is that each of these blocks aims at building a monolithic, non-modifiable application. Components, however are *deployable* entities, that is, they are not compiled into an application but are installed directly on an execution platform. This loose binding between components and their execution environments potentiates to reveal existing component structures for deleting and adding components.

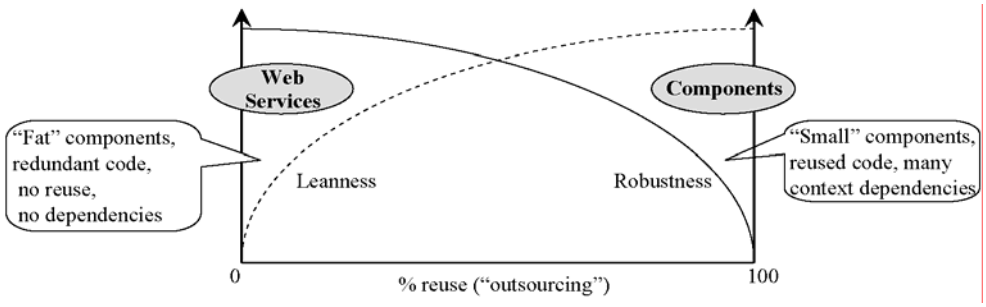


Figure 2-7: Comparison of Web services and components in the context of reuse. The decisive forces "leanness" and "robustness" are competitive values

The notion of a service as perceived from the field of SOA (section 2.3) is somewhat closer to the component approach. Some differences have to be faced, however: while components explicitly announce their necessary context dependencies (basically through the declaration of *required ports*, see next section), Web services do not explicitly reveal their dependencies to other services. In consequence, concrete implementations such as Web Services are far more self-contained, but also "fatter" than components with the risk to have much redundant code inside (Figure 2-7). According to Szyperski's analysis, self-contained services are stable against changes in their environment and maximize their use in different application contexts, but limit the degree of reuse of other services or building blocks.

Owing to their ability to declare context dependencies, components are much leaner and increase the possibility of outsourcing redundant code to third-party components (reuse!). The identifiable force fields "leanness" and "robustness" of services are competitive, however: maximizing the degree of reuse through leaner components and more dependencies minimizes the usage of components within different application contexts ("the market"). A sound trade-off between these forces has to be sought by developers and also by component vendors. Component models such as JavaBeans offer packaging mechanisms that facilitate the encompassment of basic sources (e.g. Java classes, GIF-files) together with the ability to define additional context dependencies. At this point, such trade-off solutions for Web Services and other service-oriented approaches [Cervantes and Hall, 2005] are missing.

2.5.2 Component Models

A component model conforms to a set of rules declaring standards for the implementation and the deployment of components. Usually, a component model is yielded for three target groups: the actual *component developer* and the *provider* of so-called execution infrastructures (or runtime environment) in which components are deployed later by the third target group, the *component assemblers*. Component assemblers make use of an adequate documentation in order to see how a component can be assembled with other components, as well as of a list of criteria to determine whether a component is actually appropriate.

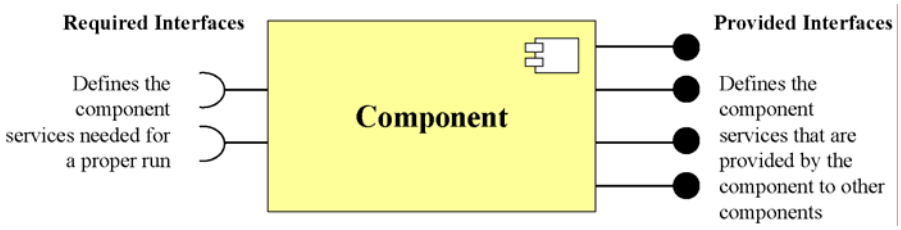


Figure 2-8: The general structure of a component

From the developer's point of view, a component model in particular specifies the interface of a component. It does so by providing exact rules for the operation names, parameter names, and types. Component interfaces can be referred to and accessed along explicit windows or so-called *ports*. In analogy to services interfaces, a single port denotes a distinguishable operation or functionality of a component. Most component models comprise two different types of ports, a (*provided) port* that specifies a *provided interface* and a (*required) port* that declares a *required interface* (see Figure 2-8). A provided interface defines the actual services (or the API) provided by the component. A required interface specifies the contextual dependencies indicating what

services by other components must be provided and then be "wired" to the interface so that the component can run properly[13]. A concrete port can be annotated by *port types* to define the valid range of values or arguments that can be passed through a port. The component model prescribes the valid interaction primitives for components, that is, the correct way how two components can communicate during runtime. For the abstract "provided-required" port concept, a number of different implementations are possible. Typical implementations are event listener, data flow, facets, as well as well as implicit dependencies denoting non-functional dependencies or requirements (such as issues concerning thread-safety or memory management). The reader should refer to [Alda *et al.*, 2002c] in order to gain an extensive overview of how these port variants can be implemented and in which component models they have already been adopted.

The most prominent and widely-discussed component models today are ENTERPRISE JAVA BEANS (EJB) [Sun, 2002], the CORBA COMPONENT MODEL (CCM) [OMG, 2002], the JAVABEANS model [Sun, 2000], and Microsoft COMMON OBJECT MODEL (COM) including its increments (for instance DCOM, or the .NET component model COM+). All these models expose different application areas, which are mostly associated to either client or server environments. EJB and CCM are designated for high end server applications with high demands on secure access to component services, persistent storage of private attributes, and transaction control on public ports. CCM has been conceived to be the standard component model for server-sided components. However, due to the long lasting standardization process entailed by the Object Management Group[14] (OMG), the conceived success and acceptance is yet unclear. For the sake of brevity, a detailed description and comparison of these models is omitted, but can be found elsewhere [Szyperski *et al.*, 2002, Marvie, 2002].

The focus of client-sided component models is to provide interfaces for building rich clients consisting of graphical user interface (GUI) components. One of the earliest models was the JAVABEANS model by Sun. JavaBeans is completely based on the Java language and encompasses the event handler model as the only way for component interaction. On top of this interaction model, basic mechanisms were introduced for component inspection. Persistence and other mechanisms known from server-sided models have not been realized. A more sophisticated standard for building rich clients has recently been established by the OSGi Alliance, the so-called OSGi Service Platform [OSGI, 2004]. OSGi aims at providing a standard component model that allows to install, update, or to remove components on the fly without ever disrupting the operation of a networked device (embedded or server). Components can thereby be discovered and dynamically bound by other components during runtime. Besides many use cases for embedded scenarios (e.g. Smart Phones), OSGi has found its way as the fundamental execution environment of the Eclipse IDE from version 3 updwards [Eclipse, 2005]. Owing to the popularity of Eclipse for professional software development, OSGi will most likely become a standard component model for rich clients.

Components are deployed in so-called execution environments (also denoted as container, application server). The purpose of an execution environment is to hide technical aspects of the underlying operating system from the components. These environments offer a couple of basic services, in particular for the deployment of components. Other services, such as transaction or persistence services, that can be used by

[13] According to Szyperski's advisement, context dependencies are even more versatile: apart from their required interfaces, components are also required to specify their needs referring to the context of composition, installation, deployment, and activation of components.

[14] see http://www.omg.org

components during run time, are also implemented by the environment itself. Component models prescribe the necessary interfaces for the respective runtime implementation. While there are a couple of open source and commercial architectures for EJB, there are up to now only few realizations for CCM. JBoss[15], for instance, is a common open source component architecture for EJB, while OpenCCM is an open source architecture for the CCM model [Marvie and Merle, 2002].

Component models are conceptualised on top of middleware technologies. The incorporation of middleware enables components to interact with other remote components across machine boundaries. Middleware hides the complexity of existing network technologies by offering a common interface for remote interaction with other hosts. CCM is designed on top of recent CORBA standards. Following these standards, recent architectures assume an ORB (Object Request Broker) as the appropriate middleware for remote interaction. EJB architectures, on the other hand, accomplish remote interaction mainly through the Remote Method Invocation (RMI) technology.

2.5.3 Component Composition

Component composition is defined as the assembly of parts (components) into a whole (a composite) without modifying the parts [Szyperski *et al.*, 2002]. Compositions of independent components are mostly formalized in a *declarative*, that is, written description. During component deployment, the execution runtime environment parses these and wires the constituting components with respect to the given assembly rules.

Since the mid-1990s, a plethora of different composition formalisms have been proposed. As the actual component approach, first preparatory work can be found several years ago. One of the earliest attempts of a composition formalism was the invention of the Module Interconnections Languages (MIL, MIL75) as proposed by DeRemer [DeRemer and Kron, 1976] [Prieto-Diaz and Neighbors, 1986]. MILs provide formal grammar constructs for identifying software system modules and for defining the interconnection specifications required to assemble a complete program. In this light, MILs are *not* concerned with what the system does, how the major parts of the system are embedded in the organization, or how the individual modules implement their functions [Prieto-Diaz and Neighbors, 1986]. Another early approach for composition-like notation has been proposed by Cremers and Hibbard [Cremers and Hibbard, 1978]. They introduced the formal notion of a *data space* that (informally) conforms to a component with an information structure (represented by cells) and built-in functions that can operate on the information structure. Data spaces can be interconnected by defining so-called *equivalences* between the cells of different data spaces.

Most composition languages found later on have been indicated as so-called Architecture Description Language (hereafter termed ADL). Predominantly, ADLs introduce the notion of a *connector* responsible for performing the interaction between the ports of components together with auxiliary services like synchronization or encryption. An ADL then defines rules stating which connectors should utilize interaction between which components. Prominent examples for connector-based languages are Darwin [Magee *et al.*, 1995] and C2 [Taylor, 1996]. In opposition to these, ADL notations like CAT [Stiemerling *et al.*, 1999] accomplish to define compositions without connectors leading to purely *connection-oriented composition* mechanisms [Szyperski *et al.*, 2002]. A circumstantial classification and comparison of existing ADLs can be

[15] see http://www.jboss.org

obtained in [Medvidovic and Taylor, 2000]. ADL are equipped with additional tools in particular to allow component assemblers to define compositions and to compile descriptions into concrete source code (C++, Java) or application skeletons (C2). For the JAVABEANS component model, the so-called Bean Markup Language (BML) has been developed [Curbera *et al.*, 2000]. BML allows to declare bindings between components (so-called *beans*) that act as a listener to events emitting from other components (sources). The actual composition and code generation (pure Java skeletons) is assisted by the BEANBUILDER tool.

ADLs are associated with a particular (or with some) architectural style(s) (see [Dustdar *et al.*, 2003], p.57 for a comparison of ADLs and their supported styles). While most ADLs realize local architectures, only few are designated for communication-based styles like client-server or peer-to-peer. ACME [Garlan *et al.*, 2000] and CAT [Stiemerling *et al.*, 1999] have been conceived for declaring distributed architectures following the client-server architectural style. Both languages allow for declaring the interfaces of client and server separately and for defining attachments (ACME) or remote bindings (CAT) between them. The inner structure of client and server can be further decomposed by means of a hierarchical component model. Apart from the formal language, both come along with tools and a runtime environment for deploying components. The runtime environment of CAT, the FREEVOLVE platform, is explained in more detail in chapter 5. Although some languages like Darwin allow for modelling peer-to-peer-like structures, no concrete language is available for modelling peer-to-peer architectures that incorporates essential concepts like advertisement of services, definition of peer groups, and so on.

Some component models fall back on the concept of contextual composition. Contextual composition deals with the automatic composition of component instances with appropriate services and resources [Szyperski *et al.*, 2002]. In this case, a component assembler can express declaratively which implicit services provided by the execution environment are needed for the proper run of components. Implicit services are services that cannot be accessed directly through public methods but are entirely managed by the execution environment. Typical examples from the EJB model are persistence, transaction, lifecycle, or security services. Components are offered to implement so-called *callback* methods in order to be notified about any state change caused by an implicit service (e.g. a method `deleteBean()` before a Bean is deleted or moved to a pool). During the execution of callback methods, the component itself is able to pursue appropriate steps before or right after a state change has occurred.

2.5.3.1 Component-oriented vs. Service-oriented Composition

A comparison between component-oriented and service-oriented composition (see section 2.3.3) clearly reveals similarities. To some extent, service composition languages like BPEL4WS can necessarily be seen as 3rd generation ADLs. Both composition types are based on the port concept to describe input and output messages which, in turn, can be grouped into higher-level (service or component) interfaces. Concrete applications (w.r.t. component-orientation) or workflow-like structures (w.r.t. service-orientation) can be composed by defining bindings between the ports of services.

A clear distinction can be made by stating that component-oriented composition declares the *structure* of an application, but makes no assumptions about control or work flows within the composition. In contrast, service composition adopts typical constructs from programming languages such as loops, conditions, sequences, or concurrency constructs to define both work and control flow between services and the local

(orchestrating) application. The inner structure of both services and local applications remains unspecified: any details of their internal implementation, their implementation platform, and infrastructures are hidden. One of the reasons for this circumstance is that WSDL (can be considered as the component (or service) model for Web Services) does not allow for the hierarchical definition of services. In this light, context dependencies on internal building blocks (e.g. database components, graphical user interfaces) within an execution environment cannot be defined.

As already stressed in section 2.4.2.1, it would be counterproductive to adopt the Web service protocol stack for generating leaner components like GUI components and for composing these with workflow-based languages. This can be justified by the complexity of both WSDL and SOAP: both exhibit a complex XML-based structure for describing even small-sized interfaces and little interactions between services. SOAP also assumes an Internet Protocol for conveying messages, which would also slow down the performance of smaller components. A suitable solution would be frameworks, such as Web Services Invocation Framework (WSIF[16]), that define a meta access protocol for accessing WSDL-based services with concrete bindings, such as Java, SOAP, or EJBs. However, the definition of bindings even for simple components certainly slows down the development process of such components.

Cervantes and Hall point out further distinctions that have to be taken into consideration ([Cervantes and Hall, 2005], p. 10). They claim that the overall focus of component-orientation is on composition, while the focus of service orientation is on discovery. Component-based applications are assembled from building blocks that are integrated at the time of assembly. In contrast, integration in service orientation occurs prior to or during execution, since only service descriptions are available during assembly. Thus, the actual services need to be discovered and eventually integrated, which leads to a shift in the integration time. Service-Orientation has to be concerned also with *dynamic availability*, that is, with the performance of adequate exception handling procedures whenever services are unavailable. Component orientation is tackled with the construction of rather static applications, whereas dynamic availability (arrival or departure of components during execution) is no real hypothesis.

The handling of exceptions within component-oriented applications is, moreover, no trivial task. The aspect of exceptions in component-based applications is discussed more accurately in the following section 2.5.3.2.

2.5.3.2 Exception Handling in Component-based Compositions

As mentioned in section 2.5.1, components should be developed as lean as possible. Also, they should expose their context dependencies to other components in order to enable the reuse of existing code (components). This pattern of decomposition, however, leads to problems with predicting the behaviour of final compositions once they have been assembled. Developers of components design them with little or no knowledge of the components with which they may interact and with no information of the context in which they will be deployed in the future. So, there is no perception of the exceptional behaviour of such assembled components or, in other words, no prediction which exceptions components should raise or how these should be handled [Alda and Cremers, 2004] [Simons and Stafford, 2004]. Exceptions might also occur due to con-

[16] See project web page for information: http://ws.apache.org/wsif/

text-sensitive interactions between components, whereas the exception originator or the exception handler cannot be assigned to a single component [Dellarocas, 1998].

The leanness of a component thus becomes a forcefield that limits the degree of handling potential exceptional cases. The leaner a component, the more accurately a developer can tailor the exception handling in a component (Figure 2-9). The problem is even increased for self-contained binary components, where code for exception handling cannot be added at a late date. The conclusion of this work is that exception handling has to be put into effect on an architectural level or on a compositional level, respectively. From there, composition languages (ADLs) and component models, including execution environments, have to be prepared to incorporate exception handling mechanisms or formalisms. In peer-to-peer-bases scenarios, such mechanisms could be adopted for handling the dynamic availability of components or services. The *component assembler* is then instructed to define mechanisms for exception handling because he has sufficient insight into the selected components of a composition.

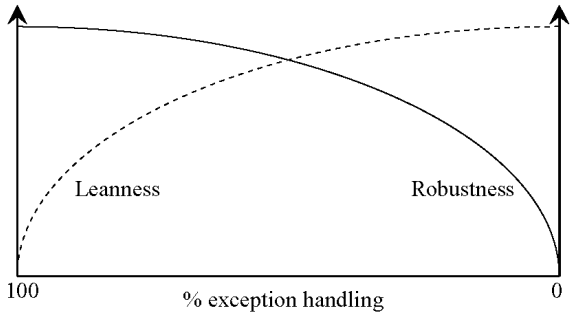


Figure 2-9: The forces "robustness" and especially "leanness" influence the degree of handling potential exceptional cases

Having analyzed existing component models and component-oriented composition languages (ADLs), this work endorses the above conclusion concerning the debate of dynamic availability and component-orientation as made by Cervantes and Hall at least to some degree. In fact, exception handling mechanisms promoting responsiveness to dynamic availability of components have hardly been realized in existing ADLs [Alda and Cremers, 2005]. Most approaches presume a stable environment without the ability to change the topology dynamically, making them less practical for dynamic architectures such as service-oriented (peer-to-peer) architectures. Although approaches like C2, CAT, and Darwin provide for dynamic modifications of compositions while the system is executing, it is not possible to define when or under what condition (e.g. because of an exception) configurations are to be carried out. The ADLs Rapide [Oreizy *et al.*, 1999] and Wright [Allen *et al.*, 1997] are the only languages that support conditional re-configuration of architectures, but in a rather restricted way. Both notations provide a where clause to determine under which conditions changes in the topology of an architectures are allowed. Components are thereby responsible for emitting special control events to trigger these changes. Externally triggered events (as known from service-oriented languages like BPEL4WS) such as onFault or onTimeout are not supported.

Exception handling on architectural level could also be realized by an implicit service that implements all necessary code for detecting and handling exceptions. Components (or composition of components) could then be linked to such a service through contextual composition (section 2.5.3). This way, the execution environment (container) would be responsible for handling exceptions. Although reasonable, this

variant cannot be found in existing (standard) component models apart from some prototypical development such as CMEH, a container-based exception handling framework for EJB containers [Simons and Stafford, 2004].

2.5.3.3 Implications for a Service-oriented Peer-to-Peer Architecture

Structural composition models (as realized by ADLs) appear more effective for the service composition model in a service-oriented peer-to-peer architecture than workflow-based composition models. This can be justified by the fact that no control structures are needed for modelling the composition of services to represent a collaboration that works towards a common goal.

Exception handling is still a crucial requirement that needs to be reflected on. Again, exception handling should occur on an architectural level rather than on a component level. As dictated by the definition of a service-oriented peer-to-peer architecture (section 2.4.3), users need to be involved explicitly during exception handling in order to handle exceptions in unforeseeable working contexts. Within the state of the art, no ADL has been found that features user integration during the process of exception handling.

2.5.4 Component-based Adaptation (Tailoring)

In this section, some general aspects of component-based adaptation methods are summarized. Subsequently, implications for adopting component-based tailoring methods into a service-oriented peer-to-peer architecture are summarized.

2.5.4.1 An Overview and Characteristics

The concept of exercising component-oriented methods as the foundation to building adaptation environments for component-based applications has initially been promoted by the work of Stiemerling [Stiemerling, 2000] and Won [Won, 2004]. The fundamental idea of *component-based adaptation methods* is to adopt typical operations for the creation of component-based applications also for the adaptation of the same. Like assembly tools that allow to compose applications, adaptation tools assume these operations as a possibility to alter the behaviour of a composed and deployed composition.

Component-based adaptation methods assume structural composition of components. Won has identified three types of adaptation mechanisms:

- Alteration of the component composition (for instance, by adding or deleting a certain component)
- Alteration of connections or links between components
- Alteration of the set of components made available for the composition

Considering the classification of adaptation strategies as elucidated in section 2.1.2.2, component-based adaptability does in particular enhance the construction of new behaviour on the basis of existing elements (according to [Henderson and Kyng, 1991]) or the integration of new functionality (with respect to [Morch, 1995]). Component-based adaptation methods also enhance the customization of single components by alternating their public parameters (e.g. to change the skin of a visual component).

These adaptation methods have been described in terms of an Application Programmable Interface (API), the so-called Tailoring API. This interface has eventually become part of the FREEVOLVE platform [Stiemerling *et al.*, 1999]. FREEVOLVE is an

execution environment for deploying component-based client-server applications. The Tailoring API has been implemented by various research prototypes (e.g. TailoringClient [Krüger, 2002]). These tools enable client owners to adapt their client-sided applications according to individual needs. Evaluation studies have shown that all these adaptation mechanisms have been perceived as intuitive and easy to learn, especially by end users. Component-based adaptation environments have therefore turned out to be adequate for supporting end-user tailoring activities.

The notion of component-based adaptability can, without doubt, be adopted for realizing an adaptive execution environment. This way, it is not the users who provide the stimuli for pursuing adaptation steps, but the system itself is responsible for carrying out these steps automatically in response to incidents occurred. The system then falls back on the same adaptation methods as originally devised for the user-driven tailoring environment. As outlined in section 2.1.2.3, both adaptive systems and systems for handling exceptions are based on the assumption to react to external incidents and eventually to compensate these through adequate handling mechanisms. In this light, component-based methods could also be considered as a fundamental methodology for handling exceptions. The work of Stiemerling and Won certainly provides a sound foundation for handling exceptions. However, concerns, like the detection of exceptions, or the handling of exceptions have not been paid any attention in their works.

2.5.4.2 Implications for a Service-oriented Peer-to-Peer Architecture

The basic component-based tailoring methods can be adopted for restructuring not only single services, but also service compositions. For tailoring service compositions, appropriate methods need to be refined, such as “addService”, “deleteService”, or addBindingBetweenServicePorts”. The intuition of these methods would clearly remain the same. Besides these obvious methods, further methods must be involved, for instance, for discovering or publishing a peer service. It could also be useful to define operations for subscribing to a provider peer. All these extra methods need to be conceived accurately.

The underlying model of a structural composition fits well for having a tailoring environment, as very intuitive operations can be defined based on the fundamental elements of the composition (i.e. components and bindings). Tailoring operations for service-oriented composition languages featuring control structures (e.g. while) require a fairly deeper understanding of programming. This is another argument for choosing an ADL as the composition language.

A further challenge results for the case that a public, component-based service has dependent consumer services, especially in an orchestration composition (section 2.3.3). As opposed to the client tailoring environment in FREEVOLVE, a peer adaptation environment will then have to take these explicit dependencies to other third-party consuming peers into consideration. An unheralded or arbitrary adaptation step could then lead to functional misbehaviours of dependent peers. Dependency management for controlling the adaptation of a local composition has not been investigated in the current work of Stiemerling and Won. For a service-oriented peer-to-peer architecture, mechanisms for dependency management are absolutely necessary. Owing to the presence of users in a service-oriented peer-to-peer architecture, users (i.e. service providers and consumers) have to negotiate in advance how such cases need to be tackled. Again, no work is known in the area of SOA and P2P for regulating the adaptation of services that could be adopted. Here, entirely new contributions have to be developed.

Tailoring methods for a service-oriented architecture have to fulfil the two most obvious requirements for tailoring routines (see section 2.1.2.2):

- *Runtime*: Executing a tailoring action should affect a service or a composition at runtime
- *Complexity*: Tailoring mechanisms should be provided at various levels of complexity so that many user groups with different skills can be supported.

Owing to the fact that these tailoring methods apply at runtime of a service composition, these methods could also be used for restructuring a composition after the occurrence of an exception (e.g. the loss of a service). Apart from these methods for handling exceptions, other mechanisms must be involved for detecting exceptions.

2.6 Conclusion

This chapter has initially proposed the approach of a service-oriented peer-to-peer architecture. This type of architecture is suitable for the development of distributed software systems (groupware) that aim at supporting dispersed working groups. It integrates fundamental aspects from two well-known architectures, namely peer-to-peer architecture and service-oriented architecture. Owing to both the omnipresence of users and the expected dynamic behaviour of the architecture, this work strives for user involvement especially during runtime of the architecture. Important tasks include the adaptation of services and service compositions, as well as exception handling. This work also proposes component-based adaptation methods in order to adapt services and compositions. By means of the component-based approach, the structure of a peer service becomes explicit. In addition, any dependency on other (internal or external) peer services can be traced. This is helpful when adapting peer services that hold dependencies to other peer services. Dependency management is a crucial requirement that the architecture has to fulfil.

2.7 Next Steps

As a next step, this dissertation presents a formal notation of an architectural style ($SO_{P2P}A$) that specifies rigorously the relevant building blocks for a service-oriented peer-to-peer architectural style (e.g. components, peer services, peers, or peer groups). Moreover, important concepts like adaptation methods, exception handling, and dependency management are introduced. The purpose of this formal architectural style is to clarify the exact operational semantics of all building blocks and, thus, of the entire architecture. The focus is set on the definition of peer services, on how they can be deployed, and on how they can be adapted. Aspects such as exception handling, integrity constraints, and dependency management are only introduced briefly.

After having shown the formal architectural style, an “instance” of that style is described in chapters 6, 7, and 8. This instance (DEEVOLVE) refers to a concrete implementation of a service-oriented architecture. In this architecture, concepts like exception handling and dependency management are treated in great length.

Chapter 3
Formalization of the Architectural Style for Service-Oriented Peer-to-Peer Architectures ($SO_{P2P}A$)

The goal of this chapter is to formalize the architectural style of service-oriented peer-to-peer architectures (hereafter superbly abbreviated as $SO_{P2P}A$) with respect to the requirements that have been elicitated in the previous chapter. The formalization proposed here is based on the *pi-calculus*, a process calculus for modeling concurrent processes. Before presenting the formalization of the architecture style in broad detail, a short introduction to the pi-calculus is provided. Afterwards, the basic elements of $SO_{P2P}A$ are outlined. More specific elements (i.e. for the adaptation of process structures as well as for handling exceptional cases) are depicted in chapter 4. The next chapter will also present an overview of related work.

3.1 The pi-calculus

The pi-calculus (also termed as π-calculus) of Robin Milner is a way of describing and analyzing systems consisting of agents (or processes) which interact with each other, and whose configuration or neighborhood is continually changing [Milner, 1991], [Milner, 1999]. Such dynamic processes are also called *mobile systems* (or mobile processes). The pi-calculus belongs to the family of *process calculi.* One of the key characteristics of process calculi constitutes the explicit modeling of concurrency issues, which is obviously the outstanding difference when compared with procedural calculi such as the famous lambda-calculus (also written λ-calculus) [Church, 1941]. Despite the demonstrable expressivity (it is Turing complete), the lambda-calculus provides no direct representation for interacting processes that are able to maintain a state over several computational steps.

Both pi-calculus and lambda-calculus exhibit commonalities due to their minimal number of definitions and their intuitive semantics. In the original lambda-calculus, everything is a function. Essentially, each function obtains an input value that yields by applying the function an output value. Even numbers are encoded as special functions that can be interrogated (by applying them) to find out which number they represent. Function application is the only observable means of computation. In the pi-calculus, every expression denotes a process, a freestanding computational activity, running in parallel with other processes and possibly containing many independent sub-processes. Two processes can interact by exchanging a message on a dedicated

named channel. More exactly, two channels can convey messages only if they have the same name. Communication along channels constitutes the only way of computation, just like a function application in the lambda calculus. A more detailed explanation of the pi-calculus is not presented in the core work but can be studied in Appendix A.

3.2 Justification for Applying the pi-calculus

The claim of this dissertation is that the pi-calculus is a suitable notation for modeling and for reasoning about dynamic service-oriented peer-to-peer architectures that incorporate the component technology as a foundation for composing and adapting application in such architectures. Obviously, a couple of similarities can be exposed when considering peer-to-peer architectures (or systems) and mobile systems that can be described by the pi-calculus. A peer-to-peer architecture consists of a number of concurrently running peers (which can be interpreted as processes) within a universal space (network). Peers do not act in isolation but collaborate with each other. In analogy to the notion of mobile processes, the configuration between these peers is not fixed but can change dynamically due to the uncertain availability of peers. Hence, the pi-calculus qualifies as a good candidate notation for modeling an architectural style that defines the rules for such dynamic peer-to-peer architectures. Reduction rules that describe the operational semantics for peer interaction have to take into account the presence of exceptions such as unavailable peers.

Another reason for adopting the pi-calculus lies in its compositionality capability: peers can be modeled as a composition of different (public) peer services that in turn can be further broken down to composition of components. On each level, the interaction principles remain consistent: channel communication (here indicated as *port* interaction) is used for modeling communication between peer services and component.

A further essential part that is specified in this section constitutes the introduction of component-based adaptation methods. These methods are applied to alter the composition of components for instance by changing connections between two components or by adding or deleting single components. As stated by Pahl, there are obvious similarities between the notions of *component evolution* (i.e. adapting the composition of components) and mobile processes as promoted by the pi-calculus [Milner, 1999] [Pahl, 2001]. Component adaptation (evolution) is also about the change of connections between components. So, the pi-calculus also appears as a suitable model for modeling component-based adaptation methods. Reduction rules can thereby be adopted for describing the operational semantics for adaptation methods, that is, the conditions when adaptations to components can be carried out without violating existing dependencies to other components (deployed by consumer peers).

3.3 The Core Elements of $SO_{P2P}A$

The following sections present the formalization of the $SO_{P2P}A$ architectural style. At first, only the core elements (component model, service model, service composition, peer and peer group infrastructure) are elaborated. More specific style aspects (adaptation, exception handling) will be illustrated in chapter 4.

3.3.1 Principle Overview

In this section, a principle overview on the $SO_{P2P}A$ architectural style is given without stressing the concrete syntax of the style. The notions of a process and that of a port are introduced by means of striking examples. The concrete syntax will be introduced in the following section.

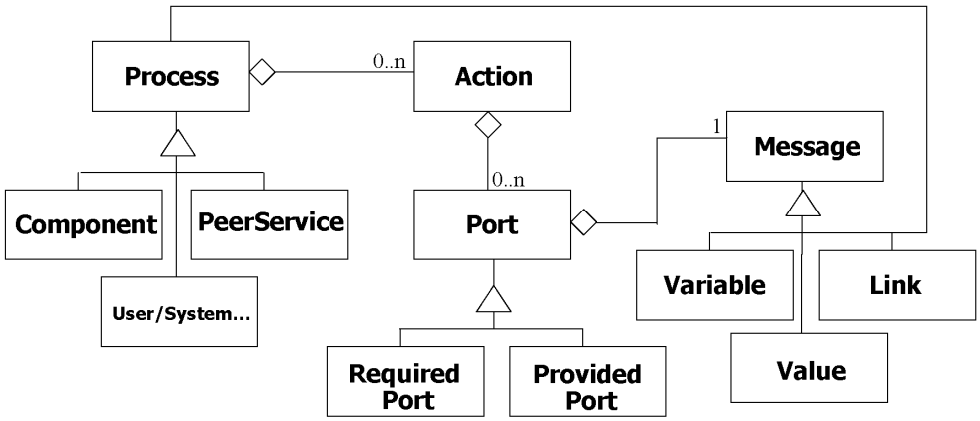


Figure 3-1: Meta-model of $SO_{P2P}A$ architectural style depicted as a class diagram

The Notion of a Process

The $SO_{P2P}A$ architectural style assumes that a distributed architecture is modeled by a set of independent processes. A process is arranged according to the meta-model given in Figure 3-1. A *process* is an autonomous entity consisting of many internal *actions*. An action realizes some behavior within a process (all valid actions are outlined shortly). An action consists of ports. A *port* is an explicit window in an action through which it can interact with actions of other processes. Each port is indicated by a *port name*. An action may consist of *required* and *provided ports.* A required port represents an interface for an action to interact with further external actions of other processes. A provided port realizes an interface to an action implementing behavior that can be invoked by an external action of a process.

The architectural style imposes a message-based interaction model. Actions of processes thereby interact with each other by sending or receiving *messages* along their dedicated ports. An action of a process, thus, accords to sending a message (output action) or receiving a message (input action). A third action corresponds to an unobservable action where no interaction takes place. An output action is typically performed along a required port, while an input action is carried out along provided port. The concept of a process, as outlined until stage, can be visualized in both a graphical and in an algebraic notation (see Figure 3-2).

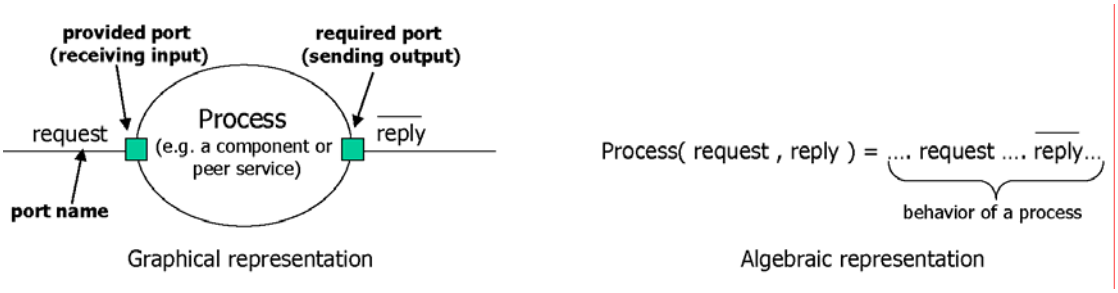


Figure 3-2: Graphical and algebraic visualization of a process in $SO_{P2P}A$

Figure 3-2 represents a process that *provides* some behavior that can be accessed along the provided port "request". After handling a request internally, the process is able to send back the result to the requesting process along required port "reply". In both representations, the port name of a required port is annotated with an overbar. A provided port has no overbar. By using the required port, an external (client) process can initiate an input action within the depicted process. In turn, sending back a result along the required port, an output action is initiated within the process. The messages sent between ports are not specified (can actually be omitted in $SO_{P2P}A$). Though ports are associated to actions, processes can also be characterized by their (public) ports. In the algebraic representation, the term *Process(request, reply)* describes a process consisting of the ports *request* and *reply*. In order to denote a process, it is also possible to omit the parentheses and port indicators (*Process*).

In the $SO_{P2P}A$ style, a process can have various concrete specializations. Typical examples for a process in this style are components, peer services, or service compositions, but also a system or a user process. This work introduces a *type system* in order to distinguish between these processes. This allows for declaring individual interaction patterns between processes of a dedicated process type.

The behavior of a process can be expressed precisely by an *algebraic expression*. Action expressions incorporate port names and messages (see later on). The behavior of a process will be formalized in a number of different ways, including sequential, parallel, and alternative behavior. Sequential behavior can be expressed by operator "." (dot). The equation

$$Component(request, reply, halt) = \overline{request}.reply.\overline{halt}.0$$

points out that process *Component* first issues a request to some component. It then waits for a reply by another component. After the reply, the process issues a halt command (along port halt). The special behavior "0" (zero) represents a terminated process. This process actually realizes the counterpart of the process in Figure 3-2. It denotes a (client) process requesting the execution of some behavior along port request. This can also be observed by the fact that a required port is formulated before the provided port (unlike in Figure 3-2). The first action in a behavioral expression is indicated as the *prefix* of a process:

$$\underbrace{\overline{request}}_{prefix}.reply.\overline{halt}.0$$

In the $SO_{P2P}A$ style, a peer service is a process that is composed out of many *parallel* processes (representing components). In order to allow for such process constructs, parallel composition is formalized by the "|"operator. The equation

$$Composition = (Component1 \mid Component2)$$

specifies a process *Composition* that is composed out of two parallel processes *Component1* and *Component2*. Peer services can further be composed towards more complex process structure, so-called service compositions. Further *process types* will be introduced such as exception handler processes or adaptation policy processes. Especially for exception handler processes, alternative behavior is important. A process that features alternative behavior is formalized by the "+" operator. The equation

$$ExceptionHandler = (HandlerAction1 + HandlerAction2)$$

states that process *ExceptionHandler* either behaves as subprocess *HandlerAction1* or *HandlerAction2*. The selection between alternative processes is non-deterministic. In $SO_{P2P}A$, a user takes over decide which alternative is to be selected (section 4.3).

Message-based Communication through Ports

In a process composition (e.g. representing a component composition), parallel processes can communicate through ports with complementary port names (i.e. one process has a provided port and the other one has a required port with the same port name). During the communication act, the required port of a process sends a *message* to the provided port of the parallel process.

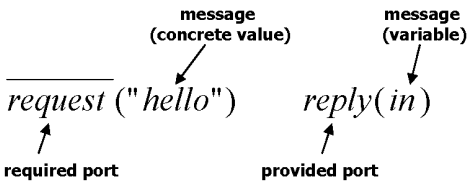


Figure 3-3: Structure of ports with messages. A message is either represented by a concrete value of by a variable

In an algebraic expression of a process, a message can transport a concrete data value (e.g. “hello”), by a variable, by a link, or a process (see also meta-model in Figure 3-1). A link is a reference to an active process. A link consumerates with the name of a port serving as an access point for that referenced process. A message is always suffixed to the port name (Figure 3-3).

With respect to the polyadic pi-calculus, a message *mess* might consist of a tuple of n sub messages: *mess* = (m_1,…,m_n). Each (sub) message has a dedicated *data type*. A data type defines a set of possible values a message can possess. In this style, three different data types will be formalized (base types, link types, process types).

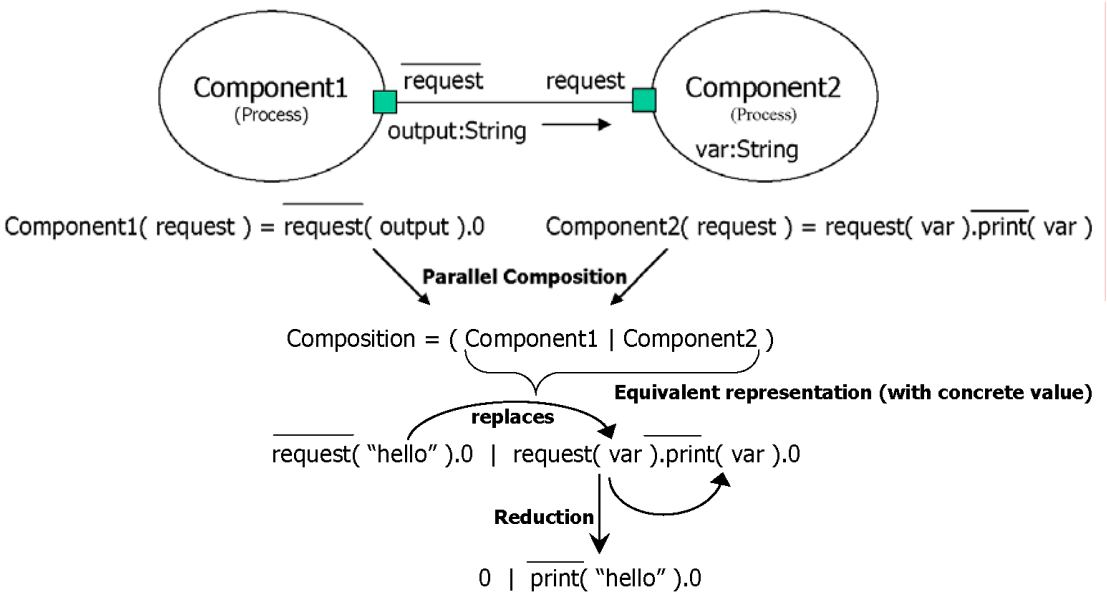


Figure 3-4: Process communication through ports in a parallel process composition. Example depicts the composition of two components

The data types of a message together with the *role type* indicating the polarity of a port makes up the *port type* of a port. Figure 3-4 illustrates an example for a port-based communication within a parallel composition. In this example, two processes (*Component1* and *Component2*) have been composed to a new process (*Composition*). In this composition, the port of *Component1* is able to send a message (*output*) to the port of *Component2*, because both have the same name (*request*) and both are type-

equal (type “String”). One can also say that both ports can *react* with each other. In a parallel composition, only the prefixes of the processes can react, that is, can exchange messages with each other. After having sent the *output* message (String “hello” in the lower process representation), the prefix of *Component1* drops out. The process terminates. At the same time, message “hello” replaces the output variable (representing an incoming message) *var* in the prefix and in the subsequent process body of *Composition2.* The prefix of *Composition2* drops out as well. Finally, process *Component2* is able of sending the received String “hello” to an imaginary process via port *print* for “printing out” the String.

In order to allow process *Component2* to receive requests by another process, it is possible to replicate the necessary actions within a process expression. This is done by using the replication operator “!”. By using this operator, an infinite number of copies of a process are assumed that can handle further request:

$$Component2 = !(request(\text{var}).\overline{\text{print}}(\text{var}))$$

The lapse of both prefixes and the replacement of messages in the receiving process is an atomic *reduction*. The reduction is the basic rule of the operational semantics of the $SO_{P2P}A$ style. Depending on the process type (i.e. component, peer service) the basic reduction rule can be augmented by a side condition in order to imply further constraints when two ports are capable of reacting with each other.

Before two processes can react with each other, port compatibility must be ensured. For doing so, both processes can define *contracts* (for peer services so-called *advertisements*) which entail concrete properties, requirements, and restrictions for reacting with a port. Contracts are made available by means of *contract ports*.

Adapting Process Structures

One of the goals of this architectural style is to allow external processes (e.g. a system process or a user process representing the interaction stemming from a real user) to adapt process structures during runtime. This can occur as a reaction on exceptional cases such as the loss of connectivity to a dependent process. In this case, component-based adaptation methods shall be used to change a given process structure in order to handle the exception. These methods merely make use of two special system ports **add** and **delete**. These ports can be part of a regular action. These ports accomplish to alter a process structure by adding or deleting further elements. Based on these rather simple operations, further more sophisticated operation can be defined.

Both ports are also used for the process of deploying processes. Besides, another system port **new** is assumed that allows for generating new instances of a process.

3.3.2 Syntax of $SO_{P2P}A$

As mentioned in the previous section, processes form the base for the $SO_{P2P}A$ architectural style. Processes are:

$$Component1, Service, Handler, Component2 \ldots \in Process$$

The identifier of the process is also referred as the name of the process (or later on, e.g., the name of the component). A name can be further indexed (e.g. $Component_i$, $Component_j^k$). Fundamental elements of processes are ports. Ports are:

$$a, b, c, aMethod, query, reply, \ldots \in Ports$$

Furthermore, messages can be conveyed through ports. Messages are:

$$message, aMessage, m \ldots \in Messages$$

Each message can consist of a tuple of sub messages:

$$message = (mess_1, mess_2, ..., mess_n) \in Messages$$

Processes can be built based on ports and messages by the syntax listed in Figure 3-5. The syntax makes use of the meta-symbol "::=" for indicating a definition.

action ::=	$\overline{port}$(*message*)	*Communication (sending via required port)*
	port(*message*)	*Communication (receiving via provided port)*
	add(*Process*)	*System port for adding a process*
	remove(*Process*)	*System port for removing a process*
	new(*Process*)	*System port for creating a new process*
	τ	*Silent Communication*
Process ::=	*Process* | *Process*	*Parallel composition*
	Process + *Process*	*Choice of processes (for user selection)*
	(v *port*) *Process*	*Restriction (local channel)*
	$\prod_{i=1}^{n} action_i Process_i$	*Sequencing actions and processes*
	! *Process*	*Replication (infinite copies)*
	{*Process*}	*Proxy to a migrated process*
	0	*Terminated process*

Figure 3-5: The syntax of the $SO_{P2P}A$ architectural style

At first, the actions of a process are declared. The communication actions for sending messages ($\overline{port}(message)$) and receiving messages ($port(message)$) can not only be used to send arbitrary messages but also to *migrate* a process into another process. This has been adopted by the higher order pi-calculus. In contrast to the original higher-order calculus, a *proxy object {Process}* to that migrated process still exists in the source process, indicated by two surrounding braces. By means of this proxy, the originating peer can still interact with the migrated process just in the same manner as interacting with a conventional process (see more information in section 3.3.4).

Besides the conventional way of passing messages between two arbitrary ports, two special *system ports* **add** and **delete** are introduced that can be used within an action. These ports are used to add or to delete a dedicated process to a given set of parallel processes. These ports are in particular used for formalizing the aspired component-based adaptation methods (see section 4.1.1). The system port **new** is used to create new processes, which is applied during the instantiation processes of components, compositions and so on. This port takes also a process as an argument. By applying this port, a copy of the passed process is generated. However, the process does not hold any state. Silent action τ does model an unobservable action used to initiate transitions without any process interaction.

Parallel composition means that a process *Process* is composed out of two concurrent sub processes (e.g. a service composition). For modelling parallel processes, the "|" operator is applied. The sum operator + models a nondeterministic choice between two processes. This operator will be used to model user decision points in particular during exception handling (section 4.3). Placing the restriction operator (*v port*) before a process expression *Process* ensures that *port* is a fresh port in *Process*. You can also think of this operator to establish a local port that is only used within a process. Given a port with the same identifier that was declared elsewhere, no confusion between messages on these two ports can occur. Sequencing indicates the sequential behaviour within a process. If an $action_i$ is used, then process *Process* behaves like process $Process_i$. Replication $!P$ means that a process is replicated an arbitrary number of times. This operator is used to replicate in particular actions providing ports in order to facilitate concurrent access. A process that is terminated is denoted as the nil (0) process. From such process, no interaction can result anymore.

In $SO_{P2P}A$, all underlying process elements representing either *deployable process constructs* (components, peer services) and *environment process constructs* (e.g. peer environment, a user interface) as well as *diagnostic concepts* (e.g. exception detection, consumer analysis) or *manipulation concepts* (e.g. adaptation methods) rest upon the *small* set of primitive notions as shown in Figure 3-5. This makes the style easy to comprehend and turns it into a *computable* calculus. In order to have a complete calculus in a mathematical sense, the operational semantics of the style yet needs to be declared, which is done in section 3.3.4. In order to describe the semantics accurately, a type system is useful to describe the different elements of the style. The type system of the proposed architectural style is formalized in the next section.

3.3.3 Type System of $SO_{P2P}A$

It was shown in the last section that message-based interaction along ports is assumed as the fundamental interaction concept for this architectural style. Ports are grouped together within processes that in turn form the basis for constructs like components or peer services. For improving the semantics of ports, messages, and processes, each of these concepts is associated by a distinguishing *type*. With respect to the fundamental work of [Cardelli and Wegner, 1985], this work interprets a type as a *set of elements* of all possible values within a universe *V*. The phrase *having a type* is then interpreted *as membership* in the respective set.

In the following, all relevant types are introduced. At first, three data types (base, process, and link) will be introduced specifying possible values that can be sent and received within a message. Then the notion of *channel type* is declared serving as way for grouping data types towards more complex type constructs. *Role type* denotes the role of a port within an interaction act. Both the channel type and the role type make up the port type of a port. Signature and predicates are special data types expressing constraints over ports. Finally, some aspects are outlined concerning subtyping.

Basic Types

Basic types are the simplest data types available. A basic type is not further specified. An example for a basic type could be an Integer, a Real, or a String type. It is assumed that at least one basic type is available. For instance, if a message *m* is of type *BASE*, then one can write *m : BASE.* In order to determine the basic type of a message, function T_{basic} can be applied. This function has the following signature:

$$T_{basic} : Message \rightarrow BasicTypeIdentifiers$$

Process Types

Process types describe sets of processes. The most trivial process type is *PROC.* Each process that can be formed by the syntax of Figure 3-5 has type *PROC*:

$$\forall\ proc \in Process : proc : PROC$$

There are several subtypes available of the general process type *PROC*. Each subtype contains special processes with special behaviour such as a component (subtype COMPONENT), a peer service (subtype PEERSERVICE), or a port expression (PORT). All subtypes of *PROC* are summarized in Figure 3-6. The dedicated type of a process can be determined by applying the auxiliary function T_{process}:

$$T_{process} : Process \rightarrow ProcessTypeIdentifiers$$

Given a process representing a component (i.e. *component : COMPONENT*), function T_{process} would deliver: T_{process}(*component*) = *COMPONENT.* In the proposed style, processes are often sent as messages, for instance, for remote deployment. Process types are not only used to express processes that can be conveyed as messages through ports but also to identify processes used for system functionality (type SYSTEMPROCESS) and for user-oriented processes (type USERPROCESS). This work makes no difference between these categories of process types.

Link Types

Link types represent ports that can be passed through an interaction between two processes, that is, between two components or services. A port serves as a link to an active process. For instance, if a distinct process needs the result of a computation from a different process, it can pass a port that directs to this process. The executing process then conveys the result back to the requesting process along this link. If a port *p* is of type *LI,* then one can also write *p : LI.* A link is also termed as a *connector* between two processes. Link types are formulated in terms of the possible data types a port can transport. The exact formulation of a link type can be reached by applying a special constructor called C_LINK. The purpose of this constructor is to classify a link based on the message that is passed along it. This way, if port *p* is of type *LI* and messages from message type *BASE* are transferred along this port, you can also write:

$$p : C\_LINK(BASE) = LI$$

The dedicated link type of a port can be determined by applying the function T_{link}:

$$T_{link} : Ports \rightarrow LinkTypeIdentifiers$$

Given a port having link type LI, function T_{link} would deliver: T_{link}(*port*) = *LI.* Basic, process, and link types can be composed to so-called *channel types*. Channel types are explained next.

Channel Type

Usually, processes such as components can use ports to send or receive an arbitrary number of messages having either basic, process, or link types. Channel types represent a coherent representation of data types that can be transported along a port. Like link types, special constructors are needed to express them. The general structure for such a constructor is indicated by the constructor C_CHAN. Assume that port *p* is of channel type *CL,* and $D_1, D_2, \ldots$ are data types (i.e. basic, process, or link data type):

$$p : C\_CHAN(D_1 \times ... \times D_n) = CL$$

In order to determine the channel type for a given port, function $T_{channel}$ can be used:

$$T_{channel} : Port \rightarrow ChannelTypeIdentifiers$$

So, $T_{channel}$ *(p) = CL.* A channel type can be compared with Records or Structs data types of imperative programming languages.

Role Type (*ROLE*)

Type *ROLE* describe the *role* of a port, that is, the conceived task or responsibility of it within an interaction. ROLE is an enumerated type that can hold a finite set of values. For instance, the enumerated types PROVIDED and REQUIRED describe the *polarity* of a port. Two ports are compatible to each other, if they have complement polarities. If port *p* has an enumerated port type *P*, one can also write *p : P*. In order to determine the role type for a given port, function T_{role} can be used:

$$T_{role} : Port \rightarrow RoleTypeIdentifiers$$

Port Type

A port type of a port is characterized by the messages it can transport (i.e. its channel type) and the dedicated role of it (i.e. role type). Therefore, a corresponding channel type and a role type construct a port type. The constructor *C_PORT* can construct a port type. Suppose that *CL* is a channel type, *p* is a port, and *ROLE* a role type:

$$p : C\_PORT(CL \times ROLE) = PT$$

PT is assumed to be the port type of p. In order to determine the role type for a given port, function T_{port} can be applied:

$$T_{port} : Port \rightarrow PortTypeIdentifiers$$

Signatures

A signature is a special type to describe the capabilities of a *port*. Signatures will be part of contracts and advertisements to describe the complete functionality of a provided port. A signature also describes the expected functionality of a required port. Signatures are produced by constructor *C_SIG*. This constructor takes all kind of data types as arguments. The syntax is as follows:

$$p : C\_SIG(D_1 \times ... \times D_n) = S$$

Function T_{sig} can be used to obtain the signature type for a given port:

$$T_{sig} : Port \rightarrow SignaturIdentifiers$$

Predicate

A predicate is a special type that represents a property or a set of properties for a *process*. An example for a property is a string that denotes the required memberships to a peer group or an integer that assesses the reliability of user. Predicates will be used for describing advertisements of peer service (processes). They are constructed by constructor *C_PRD*. It takes messages of all message types as arguments:

$$process : C\_PRD(D_1 \times ... \times D_n) = PR$$

According to this notation, *PR* is the predicate of process *p*. Function T_{prd} can be used to obtain the predicates for a given process.

$$T_{prd} : (Process \times PredicateIndex) \rightarrow PredicateIdentifiers$$

Note that a process can necessarily possess many predicates. To address selected predicates, a predicate index is assumed that uniquely identifies each available predicate.

Another helpful function can be applied on single messages in order to determine their data types (which could either be a basic, link or process type):

$$T_{dataType} : Message \rightarrow \{ProcessTypeIdentifiers, LinkTypeIdentifiers, BasicTypeIdentifiers\}$$

Predicates and signatures can be part of channel types meaning that both can be transported as a message along a port (base data types PREDICATE and SIGNATURE that can be interpreted as sub types of base type STRING). Thus, both are data types. Channel types that feature predicates and signatures are termed *special channel types*. They differ from normal channel types in how a subtype relationship can be expressed.

$PortTYPE ::=$	$C\_PORT(ROLE \times CT)$	*Port Type*
$ROLE::=$	$RoleTypeIdentifiers$	*Role Type*
$CT::=$	$C\_INTER(D \times ... \times D \times C\_REPLY) \mid$	*Channel Type of interaction port*
	$C\_CONTRACT(Q \times ... \times Q \times C\_INTER) \mid$	*Channel Type of contract port*
	$C\_ADS(Q \times ... \times Q \times C\_SUPPLY) \mid$	*Channel Type of advertisement port*
	$C\_SUPPLY(D \times ... \times D) \mid$	*Channel Type of supplier port*
	$C\_REPLY(D \times ... \times D)$	*Channel Type of reply port*
$D::=$	$BASE \mid LINK \mid P$	*Standard Data Types*
$BASE::=$	$BasicTypeIdentifiers$	*Basic Data Type*
$LINK::=$	$C\_LINK(D \times ... \times D)$	*Link Data Type*
$P::=$	$ProcessTypeIdentifiers$	*Process Data Type*
$Q::=$	$S \mid PR$	*Property Types*
$S::=$	$C\_SIG(D \times ... \times D)$	*Signature*
$PR::=$	$C\_PRD(D \times ... \times D)$	*Predicate*

Figure 3-6: The type system of $SO_{P2P}A$

Overview on the Complete Type System

Figure 3-6 summarizes the type system of $SO_{P2P}A$. The syntax uses the following meta symbols: "::=" for definition, " | " for choice (not to be mixed up with the smaller "|" operator of the syntax in Figure 3-5 denoting parallel composition), and "×" for cross product. The latter symbol is used to define the constructors for building the channel types. All constructors are explained in the next sections. Basic, link, and process data types are referred as the *standard data types*. Expressions for constructing concrete names for the type identifiers (i.e. the terminal symbols) have been omitted. All symbols used in the type system conform to non-terminal symbols.

Subtyping

Subtyping is an important feature for comparing messages having different data types and composing ports having different port types. The notion of subtyping defines rules when different types can be related to each other. It captures the intuitive idea of inclusion between types, where types are seen as collections of values. For expressing *subtype relations* in a formal way, this work refers to the formalism that can be found in ([Cardelli, 1997], section 103.6 "Subtyping"). Cardelli suggests a new judgement

$$\Gamma \mapsto A \leq B$$

stating that A is a subtype of B within a static type environment Γ. The intuition is that any element of A is also an element of B. For data and process types, this idea can be adopted (let *messA* and *messB* be messages and *procA* and *procB* be processes):

$$\Gamma \mapsto T_{dataType}(messA) \leq T_{dataType}(messB) \qquad \Gamma \mapsto T_{process}(procA) \leq T_{process}(procB)$$

For instance, the data type of message *messA* is a subtype of the data type of message *messB*, if and only if all member values of the message type of *messA* are also member values of the data type of *messB*. Formally speaking, a subtyping relation is defined as reflexive and transitive relation, each expressed by a rule. Another rule called subsumption dictates that if a term has type *A*, and *A* is subtype of *B*. then the term also has term *B* (also called the *Liskov-Substitution* rule [Liskov and Wing, 1993]). All necessary rules for message and process type are summarized in Figure 3-7.

$$\frac{\Gamma \mapsto T_{process}(p)}{\Gamma \mapsto T_{process}(p) \leq T_{process}(p)} \quad (subtyping\ reflexive, Process)$$

$$\frac{\Gamma \mapsto T_{dataType}(m)}{\Gamma \mapsto T_{dataType}(m) \leq T_{dataType}(m)} \quad (subtyping\ reflexive, Message)$$

$$\frac{\Gamma \mapsto T_{process}(a) \leq T_{process}(b) \quad \Gamma \mapsto T_{process}(b) \leq T_{process}(c)}{\Gamma \mapsto T_{process}(a) \leq T_{process}(c)} \quad (subtyping\ transitive, Process)$$

$$\frac{\Gamma \mapsto T_{dataType}(a) \leq T_{dataType}(b) \quad \Gamma \mapsto T_{dataType}(b) \leq T_{dataType}(c)}{\Gamma \mapsto T_{dataType}(a) \leq T_{dataType}(c)} \quad (subtyping\ transitive, Message)$$

$$\frac{\Gamma \mapsto a : A = T_{process}(a) \quad \Gamma \mapsto A \leq B}{\Gamma \mapsto a : B} \quad (subsumption, Process)$$

$$\frac{\Gamma \mapsto a : A = T_{dataType}(a) \quad \Gamma \mapsto A \leq B}{\Gamma \mapsto a : B} \quad (subsumption, Message)$$

$$\frac{\Gamma \mapsto \Diamond}{\Gamma \mapsto PROC} \quad (Type\ PROC, Process)$$

$$\frac{\Gamma \mapsto a : A = T_{process}(a)}{\Gamma \mapsto A \leq PROC} \quad (subtyping\ PROC, Process)$$

Figure 3-7: Basic subtyping rules for process and basic message types

The subtyping rules also introduce process type *PROC* as the base type, that is, supertype, for all processes. Hence, each process is explicitly assigned a supertype *PROC*. Following the idea of Cardelli [Cardelli, 1997], subtyping rules for channel types (in Cardelli's work represented as records) work *componentwise* and *lengthwise*. A longer channel type is a subtype of a shorter channel type. In addition, each elementary mes-

sage types of the subtype must themselves form subtypes with the elementary message types of the super type. The rule expressing the subtyping relation between two channel types can be formulated as follows (let A and B be arbitrary standard message types, CT be a channel type, m are ports):

$$\frac{\Gamma \mapsto A_1 \leq B_1 \ldots \Gamma \mapsto A_n \leq B_n \qquad \Gamma \mapsto A_{n+1} \ldots \Gamma \mapsto A_{n+x}}{\Gamma \mapsto CT_1(m_1 : A_1, \ldots, m_{n+x} : A_{n+x}) \leq CT_2(m_1 : B_1, \ldots, m_n : B_n)} \quad (\textit{subtyping channel type})$$

If two processes are willing to exchange messages along two ports, say *portA* and *portB*, then their corresponding port types must hold a *subtype* relation. This subtype relation is fulfilled if and only if:

1. the role types of these ports are complementary. This can be indicated by the *role relation* "$\cong$". If the role types of two ports are complementary, one can write:
$$T_{role}(portA) \cong T_{role}(portB)$$
2. the channel types of these ports form themselves a subtype relation ("$\leq$"):
$$\Gamma \mapsto T_{channel}(portA) \leq T_{channel}(portB)$$

The above subtyping rule can be expressed as follows:

$$\frac{\Gamma \mapsto T_{channel}(portA) \leq T_{channel}(portB) \qquad T_{role}(portA) \cong T_{role}(portB)}{\Gamma \mapsto T_{port}(portA) \leq T_{port}(portB)} \quad (\textit{subtyping port type})$$

At the moment, two ports are said as complementary if one of them is a provided port and the other one is a required port. More rules when role types are complementary will be concretized in the next sections. If channel types include constraint types (i.e. signatures and predicates), then special subtyping rules for these types must be applied. These rules will be introduced later in this chapter. All typing rules declared so far are sufficient for defining the basic operational semantics relations of the $SO_{P2P}A$ style.

3.3.4 Operational Semantics of $SO_{P2P}A$

The operational semantics of $SO_{P2P}A$ comprise two aspects. First, the communication primitives for process interacting along ports must be expressed. Here, only a single general rule can be applied for all process types. By means of a side condition, further constraints can later on be declared in order refine the general rule. Secondly, the operational semantics for process manipulations are formalized that are later on used for deploying process element as well as for adapting them.

Port Communication

The *operational semantics* of the pi-calculus is defined as a *reduction relation* $\rightarrow$ over processes. $P \rightarrow Q$ means that P can be transformed into Q by a single computational step. The *basic* reduction rule is an *axiom* that captures the ability of processes to interact through ports:

$$INTERACTION : \left(\underbrace{\overline{port}(out).A \mid port(in).B}_{P} \right) \rightarrow \left(\underbrace{A \mid [out/in]B}_{Q} \right) \langle \phi$$

The process P reduces to process Q, written $P \rightarrow Q$, if P contains two parallel sub processes that can communicate on complementary ports to become the corresponding sub process Q. These sub processes only interact between their prefixed ports. After the interaction, these prefixed ports are dropt. The data value (*out*) that is passed from the sending to the receiving port substitutes all occurrences of the formal input variable (*in*) of the preceding sub process *B* (indicated by expression *[out/in]B*).

The side condition ϕ is suitable to bring in further constraints indicating under which conditions a port interaction may be executed. The side condition basically depends on the process types of the involved processes (e.g. components, peer services) or on the data types of the messages that are conveyed along the ports. It also formulates the sub typing rules that are mandatory for a sound port interaction. The specific side conditions is formulated within the next subsections.

An important variant of the basic interaction axiom is defined for the case when processes are sent (or migrated) and receive along ports. In this particular case, a *proxy* to that process is left in the sending sub process. For that purpose, the reduction rule needs to be slightly adapted:

$$MIGRATE: \left(\overline{port}(A).P \mid port(B).Q\right) \rightarrow \left([\{A\}/A]P \mid [A/B]Q\right) \langle \phi$$

In the process Q, the migrated process *A* substitutes each occurrence of B. In the parent sub process P, the proxy process {A} replaces process *A* internally. This proxy to the migrated process *A* is later on used to interact with the (remote) process. The pertaining side condition now clarifies the constraints for this axiom:

$$\phi = \begin{cases} T_{dataType}(A) \leq PROC \wedge T_{dataType}(B) \leq PROC \\ T_{port}(\overline{port}) \leq T_{port}(port) \\ T_{process}(\{A\}) = PROC\_REF \end{cases}$$

The first constraint dictates that messages *A* and *B* need to have a process data type. Type *PROC* is the supertype of all processes, a circumstance which can be used to identify a process. The second constraints entails that both ports of the pertaining sub process need to have a subtype relation (the port type of the required port needs to be a subtype of the port type of the provided port). The last condition says that expression *{A}* – the proxy to the migrated process *A* – is of type *PROC_REF*. A process can interact with a proxy process in the same manner as with a conventional process. The following reduction rule specifies port interaction between a process and a proxy:

$$PROXY\_INT: \overline{port}(out).\{A\} \mid port(in).B \rightarrow \{A\} \mid [out/in]B \langle \phi$$

The corresponding side condition:

$$\phi = \begin{cases} T_{port}(\overline{port}) \leq T_{port}(port) \\ T_{process}(\{A\}) = PROC\_REF \end{cases}$$

Apparently, a second reduction rule must be stated for the case when *B* is the proxy process. This rule is omitted here.

Manipulation of Process Expressions

The system ports **add**, **delete**, and **new** are used to manipulate as well as to create new algebraic process expressions. The following reduction rules demonstrate the usage of these ports:

$$\textbf{add}(newProcess) \mid aProcess \rightarrow (newProcess \mid aProcess) \langle \phi$$
$$\textbf{delete}(newProcess) \mid (newProcess \mid aProcess) \rightarrow (aProcess) \langle \phi$$

The result of executing one of the ports **add** or **delete** is a manipulated process. The side condition states that all messages need to be of type *PROC*:

$$\phi = T_{process}(newProcess) \leq PROC \wedge T_{process}(aProcess) \leq PROC$$

Note that for these ports no reaction with an equal port is necessary. Process manipulation ports are later on used for formalizing adaptation methods on components and compositions. The semantics of the **new** port is as follows:

$$\textbf{new}(newProcess) \mid aProcess \rightarrow (newProcess \mid aProcess) \langle \phi$$

The side condition is equal to the previous one except of the statement that process *newProcess* does not hold any state:

$$\phi = \begin{cases} T_{process}(newProcess) \leq PROC \wedge T_{process}(aProcess) \leq PROC \\ newProcess\ holds\ no\ state\ (no\ values\ are\ in\ the\ process) \end{cases}$$

For all system ports, it is possible to pass many processes as arguments (cf. polyadic pi-calculus). This accomplishes the creation, addition, or deletion of processes in parallel. The operational semantics for *polyadic* system ports is formalized as follows:

$$\textbf{new}(process_1,..., process_n) \equiv \textbf{new}(process_1) \mid ... \mid \textbf{new}(process_n)$$
$$\textbf{add}(process_1,..., process_n) \equiv \textbf{add}(process_1) \mid ... \mid \textbf{add}(process_n)$$
$$\textbf{delete}(process_1,..., process_n) \equiv \textbf{delete}(process_1) \mid ... \mid \textbf{delete}(process_n)$$

3.3.5 Components

This section introduces the notion of a component. Components constitute the foundation for defining peer services and, above all, for the composition and adaptation of peer services and service compositions. Reviewing the state of the art on component-orientation (section 2.5) one can see that neither a clear definition and nor a manifested set of properties for a component is available. This architectural style assumes the definition and adopts *part* of the characteristics stated by Szyperski (see section 2.5.1) as its foundation for a component model. According to his work, the flexible deployment (i.e. addition and deletion) of components is one of the distinguishing characteristics of the component-oriented approach. This shall actually be the foundation for having component-based adaptation methods as stipulated in section 2.5.4.2. As an extension, however, this works assumes that these deployment operations can not only be used during start-up but also at *runtime* of a service. This satisfies the important requirement of having adaptation methods affecting service constructs at runtime (see section 2.5.4.2). Szyperski's aspired recommendation that component should not posses a persistent state is *irrelevant* for the proposed style. Components can have and can maintain an internal state if appropriate.

The architectural style furthermore adopts a message-based interaction model along ports as the core component model. This is in accordance to all major component models available nowadays (section 2.5.2). Most adaptation methods shall be declared based on these ports (e.g. changing bindings between ports) in section 4.1.1. The port model as the base for a component model is defined in the next subsection.

Public Sub Ports and their Role Types

A component consists of a number of public *component ports*. By means of these ports, a component is capable of interacting with ports of other (parallel) components. This way, (parallel) components can be placed into a composition. A component port itself is separated into a tuple of three process ports ($port_{\text{contract}}$, $port_{\text{interaction}}$, $port_{\text{reply}}$) as introduced in section 3.3.2 (see meta-model in Figure 3-8). The concept of a component port, thus, remains abstract. The task of the sub ports are elaborated next.

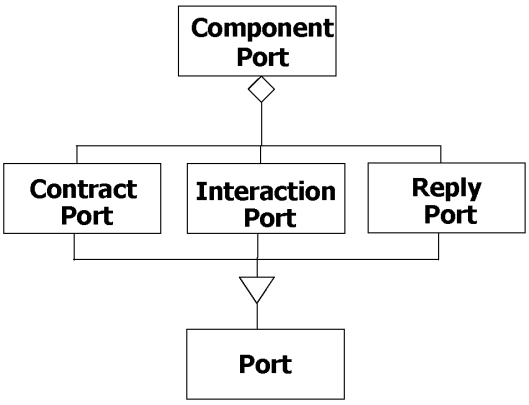


Figure 3-8: Meta-model of a component port consisting of three process ports

The first sub-port $port_{\text{contract}}$ (*contract port*) defines the contracts for a port. A contract port can either send a contract (for a required port) or receive a contract (for a provided port). A sending contract port has the following structure:

$$\overline{port_{contract}}\langle message, port_{interaction}\rangle$$

The contract sub port has a dedicated role type, called C_REQUIRED.

$$T_{role}(\overline{port_{contract}}) = C\_REQUIRED$$

A contract port of a required port specifies the interface that is expected by that port. A receiving contract port has the following structure:

$$port_{contract}\ (message, port_{interaction})$$

The contract sub port has a dedicated role type, called C_PROVIDED.

$$T_{role}(\ port_{contract}) = C\_PROVIDED$$

A contract port of a required port specifies the interface that is provided by that port.

Contract ports define, whether two ports can be composed and can interact with each other or not. Only ports having complementary contract ports can be composed. In addition, the signatures of ports have to be equal. This will be explained in more detail in the next sub section. If both contract ports match, then *interaction sub port* being passed as an argument can be taken to establish a connection between two processes.

Sub-port $port_{\text{interaction}}$ is used to invoke the functionality that is specified by the $port_{\text{contract}}$ and implemented by that port. In analogy to the contract port, sub port $port_{\text{interaction}}$ can be associated to two different role types. The role type of an interaction port depends whether this port belongs to a required or provided port. The structure of an interaction port of a required port is as follows:

$$\overline{port_{interaction}}\langle message, port_{reply}\rangle$$

The contract sub port of a required port has a dedicated role type, called INVOKE.

$$T_{role}(\overline{port_{interaction}}) = INVOKE$$

The interaction port of a required port is used to call or invoke the behavior of a provided port. The structure of an interaction port of a provided port is as follows:

$$port_{interaction}\langle message, port_{reply}\rangle$$

The contract sub port of a provided port has a dedicated role type, called EXECUTE.

$$T_{role}(port_{interaction}) = EXECUTE$$

. A component receiving a request to invoke the behaviour of its provided port specifies the sub-port of type EXECUTE. Apart from the input message, the invoking port passes a port called $port_{\text{reply}}$. This port is used to send back the results of the computation. So, for communicating the result between two components, a sub-port *reply port* is supplied with two different port types. The reply sub port $port_{\text{reply}}$ of provided port has the structure and role type as follows:

$$\overline{port_{reply}}\langle message\rangle,\ T_{role}(\overline{port_{reply}}) = REPLY$$

The structure of a reply port and its corresponding role type of required port is structure in the following manner:

$$port_{reply}\langle message\rangle,\ T_{role}(port_{reply}) = RESULT$$

The argument *message* denotes the return value that is computed.

Channel Types of Sub Ports

Until now, no clear assumptions have been made for declaring what kind of data is actually transferred through (sub) ports of a component. This can be specified by defining the channel types of all relevant sub ports. For convenience, it is assumed that a port represents a function (or method) that can take various input values and computes a single output value. Each value thereby is assigned a basic data type *B*:

$$port : B_1 \times \ldots \times B_n \rightarrow B,\ port(inputMessage_1, \ldots, inputMessage_n) = outputMessage$$

The channel type of sub-port $port_{\text{interaction}}$ used to invoke a service (client view) is determined by the channel type constructor C_INTER. This constructor defines the channel type of an interaction in terms of possible *standard types* that can be passed as input and output values. Note that through the inclusion of standard types not only basic data type but also link types can be passed. This is useful, for instance, to refer to other helping ports of other components. The output value of the computation is encapsulated by a link type. This link type is constructed by the C_REPLY constructor. This constructor classifies the reply port based on the data that has to be transferred back along this port. Such a link type is also passed as an argument for the invocation of the port functionality so that the computed result (*outputMessage*) can be sent back:

$$T_{channel}(port_{interaction}) = C\_INTER(B_1, \ldots, B_n, C\_REPLY(B))$$

The sub-port for executing a port (provider view) holds the same channel type, thus: $T_{channel}(\overline{port_{interaction}}) = T_{channel}(port_{interaction})$. As pointed out above, the reply port is specified by its own channel type:

$$T_{channel}(\overline{port_{reply}}) = T_{channel}(port_{reply}) = C\_REPLY(B)$$

The channel type of the contract port is specified by the C_CONTRACT constructor. This constructor makes use of the SIG constructor that maps any type expression to a signature that clearly specifies the interface of a port. SIG is applied to identify the

signatures of sub-ports $port_{execute}$ and $port_{invocation}$. This signature is the principle message that is sent along the contract sub-port $port_{contract}$:

$$T_{channel}(\overline{port_{contract}}) = C\_CONTRACT(C\_SIG(B_1,..,B_n, C\_REPLY(B)), port_{interaction})$$

Besides the actual signature, the pertaining interaction port is also submitted. If, for instance, an external process ascertains that both signature are equal, then the same process can take the interaction ports of both the requesting and the providing port and facilitate a connection between them. This constitutes the basic idea for the binding operation introduced in the next section.

Internal Structure of a Component

The internal behavior of a component is also modeled as a process of interacting ports. These ports cannot be used by other components. However, there is no subtle differentiation of sub ports as assumed for public ports. The style also makes no assumption about how a component is internally structured or composed. It is just presumed that the internal structure of a component consists of process elements *Proc* that in turns feature arbitrary sending and receiving ports. A sending port is modeled as follows:

$$\overline{port}\,(message),\ T_{role}(\overline{port}) = SENDER$$

A receiving part is defined like this:

$$port(message),\ T_{role}(port) = RECEIVER$$

Definition of a Component

Having exactly specified the notion of a port and its corresponding types, the definition of a component can be formulated. From a syntactical point of view, a component is characterized by an interface. An interface consists of provided and required services, which are modeled as ports. The term component refers to the specification from which concrete instances can later be generated and deployed. According to the object-oriented approach, such a template can also be denoted as a class, while the instances of a class correspond to objects. In most component-oriented terminologies, however, the distinction between component class and component instances class is omitted (see e.g. [Szyperski *et al.*, 2002]). In this work, a component indicates a template or class, while a component instance (see definition below) denotes an instance of a component. A component (template) can now be defined as follows.

Definition 3-1 (Component). Let $R = \{r_1,\ldots,r_n\}$ be a set of required component ports with $r_i = (r^i_{contract}, r^i_{interaction}, r^i_{reply})$, $P = \{p_1,\ldots,p_m\}$ a set of provided component ports $p_i = (p^i_{contract}, p^i_{interaction}, p^i_{reply})$. Let $B = \{Proc_1,\ldots, Proc_n\}$ be a set of *n* arbitrary process expressions following the syntax of Figure 3-5. Then the tuple $C = (R, P, B)$ is a component if the following properties hold:

$$i.)\forall\ r_i \in R : T_{role}(r^i_{contract}) = C\_REQUIRED$$
$$ii.)\forall\ r_i \in R : T_{role}(r^i_{interaction}) = INVOKE$$
$$iii.)\forall\ r_i \in R : T_{role}(r^i_{reply}) = RESULT$$
$$iv)\forall\ p_i \in P : T_{role}(p^i_{contract}) = C\_PROVIDED$$
$$v.)\forall\ p_i \in P : T_{role}(p^i_{interaction}) = EXECUTE$$
$$vi.)\forall\ p_i \in P : T_{role}(p^i_{reply}) = REPLY$$
$$vii.)all\ \ port\ names\ are\ unique$$

An individual port of a component *C* is referred as C.*port.*

□

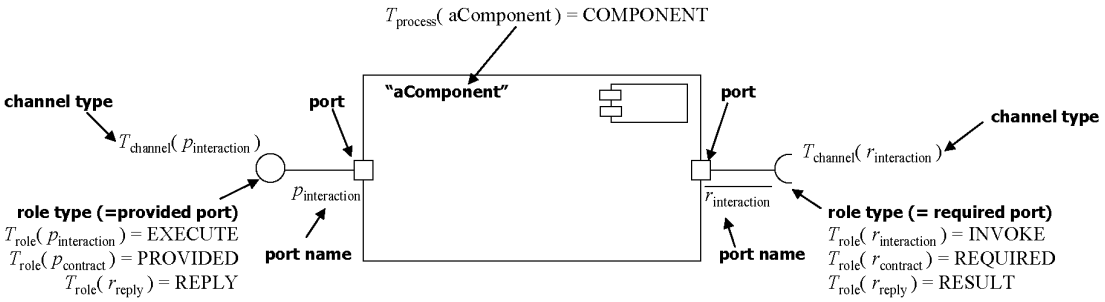


Figure 3-9: Visualization of a component in $SO_{P2P}A$ (UML component diagram). Its channel and role type constructs the port type of a port

Figure 3-9 visualizes a component and its corresponding ports based on UML's component diagram. The process of a component is arranged according to Definition 3-2:

Definition 3-2 (Component Process). Let *C = (R, P, B)* be a component. The process of a component is arranged as follows:

$$ComponentProcess = Portprocesses \,|\, Reflectionprocess$$
$$Portprocesses = ProvidedPorts \,\big|\, RequiredPorts$$
$$RequiredPorts = rp_1 \,|\, rp_2 \,|\, ... \,|\, rp_n$$
$$ProvidedPorts = pp_1 \,|\, pp_2 \,|\, ... \,|\, pp_n$$
$$rp_i = !\left(\overline{r^i_{contract}}.!\left(Proc_i.\overline{r^i_{interaction}}.r^i_{reply}.Proc_{i+j}\right)\right)$$
$$pp_i = !\left(p^i_{contract}.!\left(p^i_{interaction}.Proc_{i+j}.\overline{p^i_{reply}}.0\right)\right)$$

The *Reflectionprocess* is used to adapt the number of ports within a component and to set or delete reference to other (required or provided) ports. It makes use of the system manipulation ports **add** and **delete.**

$$Reflectionprocess = !addPort(port).\mathbf{add}(port)$$
$$|\,!deletePort(port).\mathbf{delete}(port)$$
$$|\,!setReqReference_i(channel_{x,i}, p^i_{interaction}).(\{channel_{x,i} / p^i_{interaction}\}pp_i)$$
$$|\,!setProvReference_i(channel_{i,x}, r^i_{interaction}).(\{channel_{i,x} / r^i_{interaction}\}rp_i)$$
$$|\,!deleteReqReference_i(channel_{x,i}).\mathbf{delete}(Proc_i.\overline{channel_{x,i}}.r^i_{reply}.Proc_{i+j})$$
$$|\,!deleteProvReference_i(channel_{i,x}).\mathbf{delete}(channel_{i,x}.Proc_i.\overline{r^i_{reply}}.Proc_{i+j})$$

The process type of process *ComponentProcess* depends on the *viewpoint* of the component. If the template of a component is to be specified, then the process type is:

$$T_{process}(compoSpec_x) = COMP\_SPEC$$

If the process refers to the instance of a component, then the process type is:

$$T_{process}(compoInst_x) = COMPONENT$$

The process type for a port process is independent from the viewpoint as it always depicts the specification of a process:

$$T_{process}(rp_i) = T_{process}(pp_i) = T_{process}(port) = PORT\_SPEC$$

□

The application of the standard port **new** creates a new *instance* of a component. By invoking that port, a *fresh copy* of the component process is generated that holds no state. This command represents the process of resource allocation for a component instance. Any kind of memory issues, instance administration and so forth are not concerned. The semantics of creating a component instance is as follows:

$$\textbf{new}(compoSpec_x) \rightarrow (component_x) \langle \phi$$
$$where\ \phi = T_{process}(component_x) = COMPONENT$$
$$T_{process}(compoSpec_x) = COMP\_SPEC$$

An individual port of an instance of component C is referred as C.*port.* Therefore, no distinction is made for addressing the port of a component instance or specification.

The *Reflectionprocess* can be applied on both instance and template specification. The declaration of references to other ports can only be executed during runtime, that is, after the instantiation of a component. Ports for the addition and deletion of ports can be executed on both template and instance. The difference is that port manipulation of an instance is only temporary, that is, during the lifetime of a component instance. Also, only one instance is effected by these methods. In contrast, manipulating ports on a component template is persistent. Every change affects all current instances of that component and will affect all future instances. For each $port_i$ two methods for setting and deleting references are assumed. Any operation of a *Reflectionprocess* can be invoked as follows:

$$\overline{operation}(param).aComponent \langle \phi$$
$$where\ \phi = T_{process}(aComponent) = COMP\_SPEC, operation \notin \{setter, getter\}$$
$$\vee\ T_{process}(aComponent) = COMPONENT, all\ operations$$

Operational Semantics of inner component communication

The operational semantics for port communication in a component (process expressions $proc_i$) is modeled based on the axiom for port communication (cf. section 3.3.4):

$$COMP\_INT: \left(\overline{port}(out).Proc_i \mid port(in).Proc_j\right) \rightarrow \left(Proc_i \mid [out/in]Proc_j\right) \langle \phi$$

The pertaining side condition now clarifies the constraint for this axiom:

$$\phi = T_{port}(\overline{port}) \leq T_{port}(port)$$

The side condition entails that the port types of both interacting ports need to have a sub type relation. Recall from the sub typing rule "subtyping port type" (section 3.3.3) that a sub type relation between two ports assumes a sub type relation between the channel types and complementary role types. The concrete rules for the latter assumption have not been defined yet. The rule for internal communication is as follows:

$$(complementary\ role\ type)$$
$$\frac{T_{role}(portA) = SENDER \quad T_{role}(portB) = RECEIVER}{T_{role}(portA) \cong T_{role}(portB)}$$

Naturally, both port names must be literally equal. This rule is not modeled explicitly.

3.3.6 Component Composition

Component composition refers to the parallel composition of processes of type COMPONENT so that the public ports of these components are capable of interacting with each other. Two component ports can be connected and interact safely, if the port types of the corresponding contract sub-ports ($port_{\text{contract}}$) form a subtype relationship (i.e. the port type of the required port is a subtype of the port type of the provided port). Recall that for a subtype relation between two port types, the special channel types (i.e. channel types including signatures) of the corresponding ports must also have a subtype relation. A subtype relationship between the channel types of two contract sub-ports is given, if the signature types of corresponding interaction and reply port have a subtype relationship. To formulate an appropriate typing rule, another typing rule must be provided first expressing the subtype relation between the signature types of two ports. The subtype relation is satisfied if the signatures are *literally* equal:

(*Subtype Signature*)

$$\frac{T_{sig}(\overline{portA}) = SigA.toString() = T_{sig}(portB) = SigB.toString()}{T_{sig}(\overline{portA}) \leq T_{sig}(portB)}$$

The *toString( )* operator is not further specified. It delivers a (comparable) string representation of a signature type. If both computed strings are literally the same (verified by "=" operator), then both signature ports have a subtype relation. Based on this rule, a subtype relationship between two contract ports can be expressed:

(*Subtype Special Channel Type*)

$$\frac{T_{Sig}(port^{i}_{interaction} \times port^{i}_{reply}) \leq T_{Sig}(port^{k}_{interaction} \times port^{k}_{reply})}{T_{channel}(\overline{port^{i}_{contract}}) \leq T_{channel}(port^{k}_{contract})}$$

Apparently, the signature types for both interaction and reply ports must be evaluated separately before the subtype relation of the channel types can be guaranteed.

A further constraint for a subtype relation between contract ports is that only complement role types can be connected. The following rule reflects the idea that only provided and required contract ports can be connected:

(*complementary role type*)

$$\frac{T_{role}(\overline{port_{contract}}) = C\_REQUIRED \quad T_{role}(port_{contract}) = C\_PROVIDED}{T_{role}(\overline{port_{contract}}) \cong T_{role}(port_{contract})}$$

If both constraints are met, then the port types of the contract sub ports of a required and a provided port have a subtype relation. Given this fulfilled constraint, the corresponding interaction sub ports are allowed to interact with each other. The (sound) composition of a set of individual components can now be defined based on subtype-related bindings between the ports of the constituting components:

Definition 3-3 (**Composition**). Let $K = \{C_1,\ldots C_n\}$ be a set of sound components, $B = \{ (C_a.port_i, C_b.port_j)_1,\ldots,(C_x.port_k, C_y.port_l)_n\}$ a set that defines the n bindings between required and provided ports of the available components. Then the tuple $Co = (K, B)$ is a (sound) composition, if the following condition is fulfilled:

$$\forall (C_i.\overline{port^{i}}, C_j.port^{j}) \in K : T_{port}(\overline{port^{i}_{contract}}) \leq T_{port}(port^{j}_{contract})$$

(subtype relationship of port types)

□

The underlying components are composed as a parallel process (Definition 3-4).

Definition 3-4 (**Composition Process**). Let *Co = (K, B)* be a composition. The corresponding process is to be modeled in that manner:

$$CompositionProcess = Components \,|\, Bindings \,|\, Reflectionprocess$$

$$Components = C_1 \,|\, C_2 \,|\, ... \,|\, C_n$$

The set of bindings within a composition is attached to the core process. It lists all the contract sub ports that belong to the ports to which bindings are defined:

$$Bindings = !\left(\prod_{i=1}^{n} \overline{binding_i}(C_a.port_k^{contract}, C_b.port_j^{contract}) \right)$$

The process *Reflectionprocess* allows manipulating the internal number of components and bindings of a component:

$$\begin{aligned} Reflectionprocess = {} & !addComponent(component).\mathbf{add}(component) \\ & |!deleteComponent(component).\mathbf{delete}(component) \\ & |!addBinding(C_a.port_i^{contract}, C_b.port_i^{contract}). \\ & \quad \mathbf{add}(binding_i(C_a.port_i^{contract}, C_b.port_i^{contract})) \\ & |!deleteBinding(C_a.port_i^{contract}, C_b.port_i^{contract}). \\ & \quad \mathbf{delete}(binding_i(C_a.port_i^{contract}, C_b.port_i^{contract})) \\ & |!bind(C_a.port_{interaction}^{x}, C_b.port_{interaction}^{y}). \\ & \quad (\nu channel_{x,y}(\overline{setReqReference_x}(channel_{x,y}, port_{interaction}^{x}).C_a \mid \\ & \quad \overline{setProvReference_y}(channel_{x,y}, port_{interaction}^{y}).C_b) \\ & |!unbind(C_a.port_x, C_b.port_y).(\overline{deleteReqReference_x}(channel_{x,y}).C_a \mid \\ & \quad \overline{deleteProvReference_y}(channel_{x,y}).C_b) \end{aligned}$$

The process type of process *CompositionProcess* depends on the *viewpoint* of the composition. If the template of a component is to be specified, the process type is:

$$T_{process}(composSpec_x) = COMPOS\_SPEC$$

If the process refers to the instance of a composition, then the process type is:

$$T_{process}(compoInst_x) = COMPOSITION$$

□

A visualization of a composition is illustrated in Figure 3-10. In this figure, two components are composed towards a composition. The composition itself contains of façade ports, which will be explained later on in section 3.3.7.

In analogy to a component process, the *Reflectionsprocess* can be applied to both instance and template of a composition. However, port *bind* and *unbind* can only be used on instance. New bindings can be added to an instance as well but they only apply to the instance of that composition. Adding or deleting bindings to a composition is a permanent operation that applies to all future instances of that composition. Of course, an instance of a composition consists of instances of components.

Binding Ports

An external system process (*Bindingprocess*) that belongs to the peer environment (introduced later) requests the bindings of a composition by consecutively querying (or reacting with) all its *n* $binding_i$ ports. This gained information can be used to generate an instance of a composition together with all internal bindings between the component ports. An instance of a composition implies the instantiation of all included components. The *Bindingprocess* is able to identify these components during the reaction with the *binding* ports of a composition. For the instantiation, port **new** is used to delegate the instantiation process to all *n* concrete components of a composition *x*:

$$\mathbf{new}(composition_x) \mid 0 \rightarrow \mathbf{new}(component_1^x, ..., component_n^x) \mid 0 \langle \phi$$

$$where\ \phi = T_{process}(composition_x) = COMPOS\_SPEC,$$

$$T_{process}(component_i^x) = COMP\_SPEC, \forall\, 1 \le i \le n$$

Next, a binding operator is defined that is utilized to establish concrete bindings between instances of components. Apparently, a binding between two ports stemming from two components can be defined if the port types of their contract sub-ports match according to the typing rules as elucidated above. Before the interaction sub-ports of two ports can interact, their corresponding contract-sub ports must have been resolved. The interaction between two complementary contract ports does not occur directly within their hosting *Component* processes. Moreover, a contract port prefixed to a port process (see Definition 3-2 of a component process) reacts with its complementary contract port previously passed to the *Bindingprocess* process. This way, contract ports between two component processes can never react with each other directly. The following transition rule expresses the binding process of two ports:

$$\frac{\overline{port_{contract}^i}.C_x^{Compos} \longrightarrow C_x^{Compos} \quad port_{contract}^j.Bindingpr\text{œ}ess \longrightarrow Bindingpr\text{œ}ess \qquad port_{contract}^j.C_y^{Compos} \longrightarrow C_y^{Compos} \quad \overline{port_{contract}^i}.Bindingpr\text{œ}ess \longrightarrow Bindingpr\text{œ}ess}{BIND(C_x^{Compos}.port_{interaction}^i, C_y^{Compos}.port_{interaction}^j)} \langle \phi$$

with the side condition:

$$\phi = \begin{cases} T_{port}(\overline{port_{contract}^i}) \le T_{port}(port_{contract}^j) \\ T_{process}(C_x^{Compos}, C_y^{Compos}) = COMPONENT \\ C_x^{Compos}, C_y^{Compos}\ are\ part\ of\ process\ Compos \Rightarrow T_{process}(Compos) = COMPOS \\ T_{process}(Bindingsprocess) = SYSTEM \end{cases}$$

The side condition entails that a contract port can only react with a contract port that is part of a system process (type SYSTEM). This makes sense, because otherwise the contract ports would react immediately right after the component processes have been instantiated. By involving process *Bindingprocess*, a system can control the binding process or can even improve it (e.g. by incorporating lazy binding functions or by performing additional type checks).

If the condition of the transition rule is fulfilled (expression above the fraction line), the operator BIND is invoked that establishes a private channel between the two interaction sub-ports (passed as an argument $port_{interaction}$) by invoking the bind method of the composition's instance. For this purpose, the *v* operator of the pi-calculus is applied in the bind method to introduce a fresh and local connection between two ports:

$$BIND(C_x^{Compos}.port_{interaction}^{i}, C_x^{Compos}.port_{interaction}^{j}) \stackrel{def}{=}$$
$$\overline{bind}(C_x^{Compos}.port_{interaction}^{i}, C_y^{Compos}.port_{interaction}^{j}).Compos$$

Once a binding between ports has been established, the client component (requester) can use the obtained port to invoke the service functionality of the service component. Of course, before a proper use of a client component, *all* available required ports should be connected with adequate provided ports. This is a certainly a requirement for process *Bindingprocess*, which is, however, not further specified.

Port Interaction within a composition

The interaction between two interaction ports within a composition can be described by the a transition rule (variation to the base interaction axiom of section 3.3.4):

$$COMP\_INT:$$
$$\left(\overline{port_{interaction}^{i}}(out, portA_{reply}).A \mid port_{interaction}^{i}(in, portB_{reply}).B\right)$$
$$\rightarrow \left(A \mid [out/in, portA_{reply}/portB_{reply}]B\right)\langle \phi$$

The side condition ϕ could again ensures the sub type relation between the interaction ports. This condition is actually not necessary, if one guarantees that once the contract ports have matched successfully, no type checking is further necessary. The result of the service computation is eventually sent back through the reply port (link) that was passed with the previous port interaction request:

$$COMP\_REPLY:$$
$$\left(\overline{port_{reply}^{i}}(result).B \mid port_{reply}^{i}(in).B\right) \rightarrow \left(B \mid [result]A\right)\langle \phi$$

Again, the site condition for guaranteeing the type equality of the reply ports could be omitted. For dynamic type checking mechanisms, this would clearly be useful. Note again that a reply port is not mandatory for the proposed interaction pattern. If no result is sent back, a uni-directional interaction is assumed (event notification).

3.3.7 Peer Service

In this section, the notion of a peer service is introduced. Informally speaking, a peer service consists of two different component compositions: a *local composition part* that defines the actual implementation of the peer service as well as an *interface composition part* that declares the interface to the peer service. While the local part remains at the provider peer, the interface part is *migrated* to a consumer peer, so that it can access the peer service accordingly. To this end, further notations must be introduced for defining bindings between dedicated ports of two compositions. The set of dedicated ports is referred to as the *facet* of a composition, which will be defined in the next definition (Definition 3-5):

Definition 3-5 (Façade). Let *Co* be a composition, *P* the set of ports $\{P_1,...P_n\}$ used in this composition, and $F_{Co} \subseteq P$ a subset of these ports. The set $F_{Co} = \{F_1,...,F_n\}$ defines the façade of composition *Co*. The process of a Façade is attached to the original composition of a peer service:

$$FacadeProcess = Ports \mid Reflectionprocess$$
$$Ports = port_1 \mid ... \mid port_n$$

The process Reflectionprocess accomplishes to add and to delete single ports from the façade:

$$Reflectionprocess = !addPort(port).\mathbf{add}(port) \mid !deletePort(port).\mathbf{delete}(port)$$

An individual port of a façade is indicated as $F_{Co}.port.$

□

The shift of ports from a composition towards a façade is termed *vertical binding*. In contrast, binding ports between two equal processes (e.g. two *components*) is termed *horizontal binding*. Figure 3-10 visualizes both the composition of two components by means of horizontal bindings and the provision of facades port by vertical bindings. The visualization is based on the composite structure diagrams from the UML.

Two different compositions can be bound towards a distributed composition (only) through ports that are part of their facades. A distributed composition thus can be defined in the following way:

Definition 3-6 (Distributed Composition). Let *CoL* be a composition denoted as the *local composition part*, *CoI* a composition denoted as the *interface composition part*, F_{CoL} be the façade of *CoL*, and F_{CoI} the façade of *CoI*, and $B = \{ (F_{CoL}.port_i, F_{CoI}.port_j)_1,\ldots,(F_{CoL}.port_k, F_{CoI}.port_l)_n\}$ a set of *n* bindings between ports of the facades. Then the tuple $DC = (CoL, CoI, F_{CoL}, F_{CoI}, B)$ is a distributed composition.

□

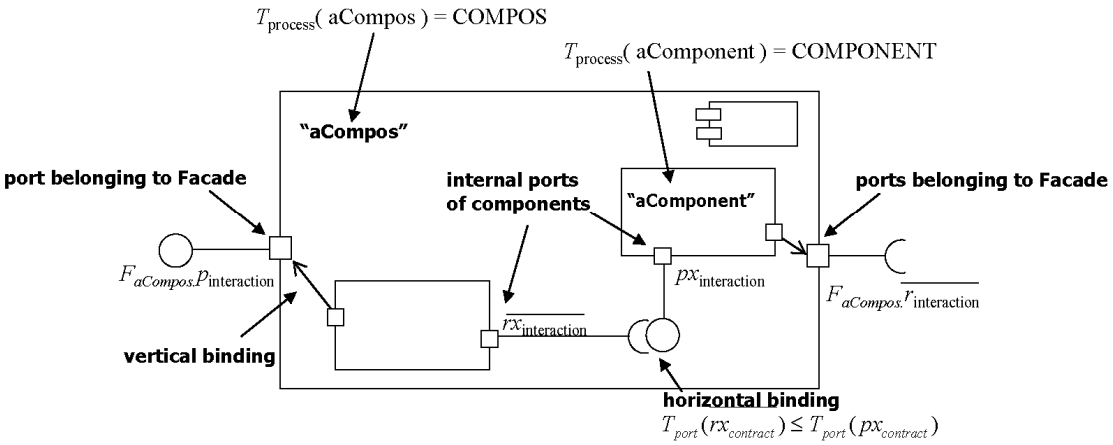


Figure 3-10: Composition of two components towards a composition. The composition is augmented by façade ports (UML composite structure diagram)

Definition 3-7 (Process of Distributed Composition). The process of a distributed composition $DC = (CoL, CoI, F_{CoL}, F_{CoI}, B)$ is defined as follows.

$$DistributedCompositionProcess = Compositions \mid Bindings \mid Reflectionprocess$$

$$Compositions = Co_{CoI} \mid Co_{CoL}$$

$$Bindings = !\left(\prod_{i=1}^{n} \overline{binding}_i (F_{CoL}.port_k^{contract}, F_{CoI}.port_j^{contract}) \right)$$

The process *Reflectionprocess* is used to alter the bindings between the façade ports belonging to the interface composition and local composition part, respectively:

$$Reflectionprocess = !addBinding(binding_i).\mathbf{add}(binding_i)$$
$$|!deleteBinding(binding_i).\mathbf{delete}(binding_i)$$
$$|!bind(F_{CoL}.port_x, F_{CoI}.port_y).(\nu\, channel_{x,y}(\overline{setReqReference_x}(channel_{x,y}).C_{comp(port_x)} |$$
$$\overline{setProvReference_y}(channel_{x,y}).C_{comp(port_x)})$$
$$|!unbind(F_{CoL}.port_x, F_{CoI}.port_y).(\overline{deleteReqReference_x}(channel_{x,y}).C_{comp(port_x)} |$$
$$\overline{deleteProvReference_y}(channel_{x,y}).C_{comp(port_x)})$$

□

It is assumed that within a distributed composition, the participating composition parts remain stable so that no operations need to be declared for adding or deleting a composition part (adaptation and modification rather occur on the levels of atomic component and composition). In the *bind* and *unbind* command, function *comp* is applied as a helping function to determine the concrete component that hosts a given façade port:

$$comp : Port \rightarrow \{ComponentTypeIdentifiers\}$$

This way, bindings between façade ports are only of virtual nature.

The instantiation of a distributed composition is achieved by delegating the creation process to the respective local and interface composition. The different creation strategies of distributed compositions are explained later in the section on creating a peer service (see section 3.3.10).

Public Interface

For the realization of a distributed composition towards a public peer service, yet some further refinements need to be done. The façades only define interfaces that are not visible for the service consumer. Another type of interface, the so-called *public interface* (API) is introduced later in order to define a publicly accessible interface for a peer service that can also be used by service consumers for their application purposes. Informally, a public interface comprises a set of (public) *facade ports*. A new sub-port type called $port_{\text{advertisement}}$ is used to obtain properties of the whole public interface. This port is attached to the process representing the API. The structure of an advertisement sub port is as follows:

$$port_{advertisement}(message), T_{role}(port_{advertisement}) = AD\_PROVIDED$$

The counterpart of this sub port does define expected properties of a public interface:

$$\overline{port_{advertisement}}(message), T_{role}(\overline{port_{advertisement}}) = AD\_REQUIRED$$

The architectural style makes no assumption concerning the location of this sub port. This could be a service that offers extension points to other peer services. Also, a user or system process could define properties for a desired peer service.

If two advertisement ports should match (i.e. the provided and expected properties are the same), complement port types are required. Beyond complementary role types of the respective ports, a subtype relation of the channel type is presumed. The special channel type of sub-port $port_{\text{advertisement}}$ consists of more conditions than $port_{\text{contract}}$. In particular, an *advertisement message* includes a set of predicates that describe properties. Informally speaking, an advertisement contains the following property fields:

- A property *Sem* indicates a semantical properties of the interface of a provided service. This condition is rather general-purpose and not further specified. It could, for instance, comprise a pre- and post condition that must be fulfilled before (or af-

ter) the invocation of a service port. This would lead to design by contract rules for ensuring the compatibility between services. Another option would be to enhance semantic description of the service by means of an *ontology*. An ontology would clearly specify the meaning of a service by means of concepts-relationship modelling approach [Radetzki and Cremers, 2004].

- An advertisement also prescribes the memberships *Mem* of one or more groups a potential consumer peer must belong to. The notion of peer groups will be described later on.
- The property *User* could contain of user- or peer-specific data such as reputation values of service providers. Such values could be expected especially by service consumers, so that they can assess the trust of a provider, for instance, in terms of availability, performance, or the frequency of controlled adaptations.
- The attribute *SIG* represents the actual interface of the service. This includes all required and provided ports.
- A unique service identifier (*sid*) to uniquely identify a service.
- *C_SUPPLY* is the constructor defining the channel type of the so-called *supply port*. This port is later on used for exchanging all necessary data required for the deployment and thus for the execution of a peer service. It is only used after the properties of a peer service match the required ones of an advertisement.

A peer willing to bind a provided peer service must fulfil the properties implied by the advertisement of that service. On the other hand, the provided peer service must also fulfil some properties that are implied by the consumer (e.g. the reputation of a provider). An advertisement that is later on used for describing the interface of a peer service is termed *peer service advertisement* and is defined as follows:

Definition 3-8 (Peer Service Advertisement). Let *ads = (sem, mem, user, sig, sid, supply)* be a message. Message *ads* is said to be a peer service advertisement, if the following conditions for its sub messages are fulfilled:

- $T_{\text{dataType}}(\ sem\ )$ = STRING, where *sem* is a semantic property specifying the interface of a peer service. In the easiest way, this could be an unstructured text. The predicate type of this property is *C_PRD(sem) = SEMP_PROP.*
- $T_{\text{dataType}}(\ mem\ )$ = STRING, where *mem* is a semantic property specifying the group memberships of a peer service. *C_PRD(sem) = MEM.*
- $T_{\text{dataType}}(\ user\ )$ = INTEGER, where *user* is a semantic property specifying the current reputation of a user. *C_PRD(sem) = USER.*
- $T_{\text{dataType}}(\ sig\ )$ = SIGNATURE, where *sig* is a semantic property specifying the signature of the ports of the API.
- $T_{\text{dataType}}(\ sid\ )$ = INTEGER, where *sid* is the unique identifier of the service
- $T_{\text{dataType}}(\ supply\ )$ = LINK, where *supply* is a link to the supply port of a provider peer, where all necessary information for the deployment of a peer service can be obtained.

The data type of ads is SERVICE_ADS: $T_{\text{dataType}}(\ ads\ )$ = SERVICE_ADS. This data type is a specialization of the (base) data type PREDICATE. All specified properties are put together into an advertisement by constructor C_ADS:

$$C\_ADS(\ C\_PRD(Sem, Mem, User, SIG(D \times ... \times D)), sid, C\_SUPPLY(D \times ... \times D)\ )$$

This constructor is used for generating a channel type and must therefore be assigned to a concrete port. This is done in the definition of a public interface:

Definition 3-9 (Public Interface). Let $DC = (CoL, CoI, F_{CoL}, F_{CoI}, B)$ be a distributed composition, $A = \{a_1,\ldots,a_n\}$ with $A \subseteq F_{CoI} \cup F_{CoL}$ be a set of ports. Let *ads* be a peer service advertisement, that is, $T_{dataType}(ads)$ = SERVICE_ADS. Then, the tuple $API = (A, ads)$ is a public interface of a distributed composition**.**

The process of a public interface can be structured in this way:

$$APIProcess = Ports \mid Advertisement \mid Reflectionprocess$$
$$Ports = port_1 \mid ... \mid port_n$$
$$Advertisement = port_{advertisement}(ads)$$

The advertisement of a public interface can be obtained along the advertisement port of process *Advertisement*. The channel type of this port can be computed as follows:

$$\mathrm{T}_{channel}(\overline{port_{advertisment}}) = C\_ADS(\, C\_PRD(Sem, Mem, User,$$
$$SIG(D^1_{port1} \times ... \times D^n_{port1}), ..., SIG(D^1_{portn} \times ... \times D^n_{portn},)$$
$$sid, C\_SUPPLY(D \times ... \times D)\,)$$

The advertisement takes the signatures of *all* available ports within the API. Process *Reflectionprocess* allows to add or to delete single ports from the public API:

$$Reflectionprocess = !addPort(port).\mathbf{add}(port)$$
$$\mid !deletePort(port).\mathbf{delete}(port)$$

An individual port of a public interface is indicated as $API_{DC}.port.$

□

From the definitions of a distributed composition *DC* and that of a public interface *API*, a definition for a peer service can directly be derived:

Definition 3-10 (Peer Service). Let $DC = (CoL, CoI, F_{CoL}, F_{CoI}, B)$ be a distributed composition and API_{DC} = (*A, ads*) the public interface of the distributed composition *DC*, and *sid* a universally unique integer, $T_{data}(sid)$ = INTEGER. Then the tuple *PS* = (DC, API_{DC}, *sid*) is a peer service.

□

The process of a peer service is that of a distributed application with a dedicated public interface.

Definition 3-11 (Peer Service Process). The process of a peer service *PS* = (DC, API_{DC}, *sid*)) is as follows:

$$PeerServiceProcess_{sid} := DistributedCompositionprocess \mid APIprocess$$

The process type of a peer service process depends on the viewpoint. The template of a peer service has the process type:

$$T_{process}(PeerServiceProcessTempl_{sid}) = \text{SERVICE\_SPEC}$$

The instance of a peer service has the process type:

$$T_{process}(PeerServiceProcessInst_{sid}) = \text{PEERSERVICE}$$

The peer service advertisement summarizes all port interfaces and, thus, describes the complete functionality for a given peer service. Its predicates are associated to the template of a peer service process and is obtained by applying function T_{prd} . Example:

T_{prd}($PeerServiceProcessTempl_{sid}$, “mem”) = MEM

3.3.8 Peer

After having defined the notion of a peer service, further definitions can now be derived like that of a peer and of a peer group. A peer is a collection of (parallel) peer services that can be offered to other peers. As an environment for hosting peer services, a process called *Supplyprocess* is added to the peer. This process takes over any kind of behaviour necessary to carry out the deployment of services, the distribution of advertisements, or the message exchange with other peers. A peer also consists of a process termed *Userprocess* that represents an interface to a user. This process is applied whenever a user is involved or needed for decision-making (e.g. exception handling). The *Userprocess* does not model system behaviour but user behaviour. The supply port is the public default port for accessing a peer.

Definition 3-12 (Peer). Let $PS = (P_1,\ldots,P_n)$ be a set of public peer service, *Userprocess* and *Supplyprocess* two processes and $port_{supply}$ a port, and *pid* a universally unique integer. Then the tuple $PR = (PS, Userprocess, Supplyprocess, port_{supply}, pid)$ is a *peer environment* or simply a *peer*. Port $port_{supply}$ has the following types:

- $T_{\text{channel}}(port_{supply})$ = $C\_SUPPLY(D\times...\times D)$
- $T_{\text{role}}(port_{supply})$ = SUPPLY_PROVIDED

The processes have the following process types:

- T_{process}(*Userprocess*) = USERPROCESS
- T_{process}(*Supplyprocess*) = SYSTEMPROCESS

□

The process of a peer is arranged in the following way:

Definition 3-13 (Peer process). The process of a peer $PR = (PS, Userprocess, Supplyprocess, port_{supply}, pid)$ can be arranged as follows:

$$PeerProcess_{pid} = PeerServices \mid Supplyprocess \mid Userprocess \mid Methods$$
$$PeerServices = p^1_{sid} \mid p^2_{sid} \mid ... \mid p^n_{sid}$$
$$Supplyprocess = !\left(port_{supply}(method, message, reply).\overline{method}(message, reply)\right)$$
$$Methods = method_1(message, reply).B \left| ... \right| \; method_n(message, reply).B$$

The process type of process *Peerprocess* is T_{process}(*Peerprocess*) = PEER. A Peerprocess may enter either of two roles namely peer service provider or peer service consumer. Depending on that role, the process type also varies:

T_{process}(*Peerprocess*) = PEER_CONSUMER

T_{process}(*Peerprocess*) = PEER_PROVIDER

The process type of the peer services is:

T_{process}(p_{sid}) = SERVICE_SPEC

where *sid* is the unique identifier of the peer service. It is also possible to denote a peer service *sid* provided by peer *pid* with:

$$peerService_{sid}^{pid}$$

The *Supplyprocess* and the *Userprocess* could also be annotated with a substring in order to associate these processes with the given role of a peer. Examples:

$$Supplyprocess_{Provider}; Userprocess_{Consumer}$$

□

The exact and complete semantics of the *Supplyprocess* and *Userprocess* will not be provided in this work. This would actually blow up the work. If appropriate, the signatures of single methods within a *Supplyprocess* will be provided only. In fact, it is up to a peer to decide how these methods are implemented. For instance, a peer could act as a rendezvous peer that is responsible to assimilate service advertisements through a peer-to-peer network. Then, a routing method is important for this task. In contrast, a normal peer that only consumes a small number of services need not necessarily rely on supporting routing methods.

Note that the definition of a peer mainly defines the *provider role* a peer might adopt during its lifecycle. No assumptions are made, how consumed peer services are loaded, deployed, and maintained within a peer environment. These aspects are hidden within the *Supplyprocess*. You will see later in the following sections how the methods of this process could take over the task to handle the deployment of peer services.

A *peer advertisement* summarizes the descriptions of all peer services advertisements and, thus, describes the complete functionality for a given peer. A peer advertisement is not further specified.

3.3.8.1 Peer-to-Peer Architecture

The collection of all available peers is termed a peer-to-peer architecture.

Definition 3-14 (Peer-to-Peer Architecture). Let $PS = (P_1,\ldots,P_n)$ be a set of peers. That collection is termed peer-to-peer architecture.

Note that the definition makes no assumption whether peers are interconnected or not. An interconnection or a dependency may result from consuming peer services from a provider peer. Peers can also depend on rendezvous peers for querying advertisements and for receiving results upon such query. The purpose of a rendezvous peer in a peer-to-peer architecture is highlighted in the next section.

The topology of a peer-to-peer is by no means fix but change dynamically upon the failure or arrival of new peers. The leads to mechanisms for handling failed peers, which are presented in chapter 4.

3.3.8.2 Routing Mechanisms for Rendezvous Peers

Each peer may take over the role of a *rendezvous* peer. A rendezvous peer is responsible to disseminate service, peer and peer group (see 3.2.6) advertisements through a peer-to-peer architecture. Through this process of sharing of advertisements, the likelihood for finding advertisement in decentral topology is increased. Also, rendezvous peers may serve as a connection point for regular peers for looking up certain advertisements. What rendezvous peers make themselves distinguishable from regular peers is the provision of so-called *routing mechanisms*[17]. In fact, many strategies for indexing peer in a decentral topology, routing queries through an index, or updating index

[17] Other criteria such as performance, reliability, or connectivity are possible and worthwhile to consider. They will not be formalized in this stye.

structures can be found in the literature (c.f. CAN, Chord). In this style, only the relevant public methods for these mechanisms are indicated.

$$Supplyprocess_{Rendezvous} = !\left(port_{supply}(method, message, reply).\overline{method}\langle message, reply\rangle\right)$$

$$\begin{aligned} Methods = {} & publishAds(\ data, source\ ).B_{publish} \\ & \overline{forwardAds}(\ data, target\ ).B_{forward} \\ & queryAds(\ data, source\ ).B_{query} \\ & \overline{forwardQuery}(\ message, source, target\ ).B_{forward} \\ & updateRoutingTable(\ data\ ).B_{update} \end{aligned}$$

The exact specification of a strategy would certainly go beyond the scope of this work; thus these methods are only explicated reasonably. Method *publishAds* can be invoked by a regular peer to publish data on this peer. Along this port, only advertisements can be conveyed. Thus, T_{data} *(publishAds) = C_ADS(...)*. Argument *source* is a link type to the originating peer. With respect to internal preferences, a rendezvous peer can forward this advertisement to other (neighboring) rendezvous peer it is aware of. This is done by invoking method *forwardAds.* Any peer can query or discover distinct advertisements by invoking method *queryAds*. Argument data can be seen as a basic data type. A rendezvous peer can then handle this query request. If it possesses the desired advertisement, the advertisement is returned to requesting peer along the source port. A match between a query argument and an advertisement may be based on comparing the query string with the property fields of an advertisement (see [Zaremski and Wing, 1997] for possible matching algorithms). If the appropriate advertisement cannot be found, the query is forward to a set of other rendezvous peers by invoking method *forwardQuery*. This process will last until an appropriate advertisement is found.

In a dynamic network environment such as in a peer-to-peer architecture, it is necessary to become acquainted of new or disappeared peers. For the accurate dissimilation of advertisements and queries, information on available (neighboring) peers must be updated. This information is stored in a routing table. From time to time and according to global rules, this routing table can be updated through the invocation of the *updateRoutingTable* method. Strategies on how to formalize these rules are not addressed here in this work (cf. routing strategies in CAN or traditional routing algorithms such as the Distinct Vector algorithm).

3.3.8.3 Message Exchange

Any peer process should offer the ability to send message to any other identifiable peer process. This message exchange behavior can later on be used for exchanging arbitrary messages between peers. Particularly, message exchange is practical for notifying peer on upcoming events such as the adaptation of public peer services (cf. section 4.1.2) or for notifying peer on a planned service adaptation (cf. section 4.1.2). A suitable process should contain two different methods for sending and for receiving messages (*message* is the sent or received message, *target* the link to the receiving peer process, and *source* the link to the sending process):

$$Supplyprocess = !\left(port_{supply}\right) \mid Methods$$

$$\begin{aligned} Methods = {} & \overline{sendMessage}(\ message, target\ ).B_{sending} \\ & \mid receiveMessage(\ message, source\ ).B_{receiving} \end{aligned}$$

The process behavior after having sent and received a message is not formalized. Also, an error handling mechanism (e.g. for broken messages) is omitted.

3.3.9 Peer Group

A peer group is a collection of different peers within a peer-to-peer architecture sharing common topics or competences. The next definition declares a peer group:

Definition 3-15 (Peer Group). Let $PS = (P_1,\ldots,P_n)$ be a set of peers, $L = (L_1,\ldots,L_n)$ another set of peers with $L \subseteq P$, and *pgid* a unique integer. Let *PC* be a process and *ads* be a message. Then the tuple $PG = (PS, L, pgid, ads , AS)$ denotes a *peer group*.

The set *L* represents the founder of a peer group.

AP is the *adaptation policy* for a group.

PC denotes a process termed *policy checker* for evaluating the adaptation policy

□

Its associated peers mainly determine the capabilities of a peer group. A *peer group advertisement* can be utilized for describing a peer group of interests, competencies and so forth. It also summarizes the *group services* of group. Group services refer to services that are provided by the members of a peer group. At least one peer is responsible for providing these services.

Definition 3-16 (Peer Group Advertisement). Let *ads = (sem, ap, peers, services)* be a message. Message *ads* is said to be a peer group advertisement, if the following conditions for its sub messages are fulfilled:

- $T_{dataType}(sem)$ = STRING, where *sem* is a string representing a semantic property to determine competencies, interests, knowledge, or common rules of a peer group. In the easiest form, this could be an unstructured text.
- $T_{dataType}(ap)$ = STRING, where *ap* is a semantic property specifying the adaptation policy.
- $T_{dataType} (peers)$ = STRING, where *peers* is a semantic property specifying the representative peers of that group.
- $T_{dataType}$ (*services*) = SIGNATURE, where *services* is a semantic property specifying the signature of *group services* available in that group.

The data type of *ads* is GROUP_ADS: $T_{dataType}$(*ads*) = GROUP_ADS. This is specialization of the (base) data type PREDICATE.

□

It is certainly possible if not necessary to include more information in such advertisement. In this style, only those relevant aspects are formalized, other information (e.g. information concerning the funding peers) may be incorporated on demand.

3.3.9.1 Adaptation Policy

Each peer group prescribes its own policy or rules how to cope with adaptations on peer services within a given group. This way, uncontrolled or unwary adaptations by provider peers can be suppressed. Any time a peer provider is willing to adapt a service, he has to evaluate the adaptation policies of all group the peer is a member of. A policy, for instance, could dictate that no adaptation is possible if consumer dependencies exist to that service. Besides the policy, a peer is able to download the policy checker process (*PolicyCheckerprocess*) through one of the supply ports of group

founder (not further modeled). This allows evaluating whether an intended adaptation can be carried out or not. The policy checker makes use of information from the subscription process (see next section). This procedure will be elaborated in section 4.1.2.

It is further assumed that the originator of a peer group have agreed to an adaptation policy during the creation of a peer group. This process of negotiating the policy is not modeled. In addition, it should be remarked that adaptation policies may evolve during time. This might be case, if regulations or opinions concerning an existing policy alter.

3.3.9.2 Applying, Joining, and Resigning a Group

At least one peer must be responsible for the creation of a group. In the course of its lifetime, this group of peers is also concerned to maintain that new peer group. Maintenance of a group mainly comprises the application, the joining as well as the resigning of a group. For supporting this process, each peer serving as one of founder of a group must include the following methods to its supply process:

$$Supplyprocess_{Founder} = !(port_{supply})$$

$$Methods = apply(credential, reply).B_{apply}$$

$$join(credential).B_{join}$$

$$resign(credential).B_{resign}$$

The flow of these methods is as follows: a peer willing to join a group has to apply for a membership. One of the founding peers can only approve this application. For the application, a peer passes a credential as an argument. Such a credential is comparable to an application form including, for instance, the rationale for the application. This credential together with a reply port is added to an internal process that collects and evaluates all incoming application. Through this reply port, a peer is accomplished to approve or to reject the application. Given the approval of an application, the requesting peer is enabled to join to that group by invoking the join method together with the same credential (for authorization purposes). At any point in time, a peer is able to resign from a peer group by invoking method resign. Methods or rules defining the prerequisites for the acceptance of a peer are not formalized here.

3.3.10 The Deployment of a Peer Service

The invocation of a peer service is more complex than the invocation of a component service port. Before the first invocation can actually be carried out, a service needs to be created on the provider peer and deployed on consumer side. For doing so, a couple of preliminary data need to be exchanged between provider and consumer (fig. 3-11).

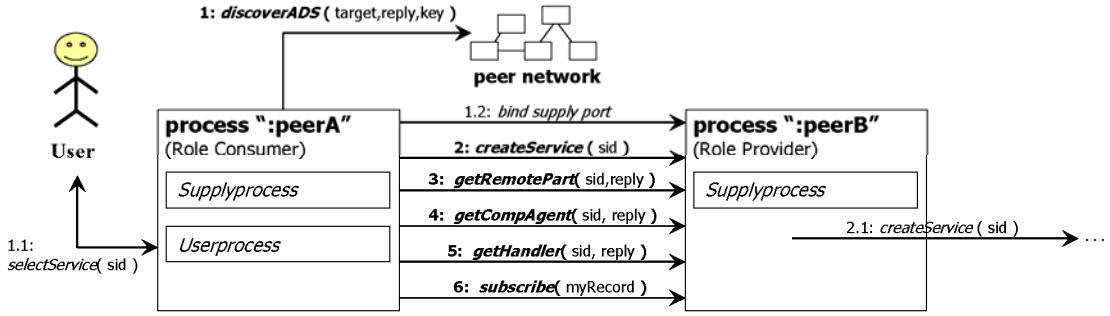


Figure 3-11: Information exchange between service consumer and provider during service deployment (based on UML communication diagram)

This entire process of creating a service and exchanging necessary data is termed *service deployment*. This process of deployment is done along the supply channels of provider and consumer peer. In contrast to all other ports modelled so far, supply ports are capable of transporting concrete processes. Such processes indicate either components or agents that operate on the behalf of a provider peer.

There are six relevant operations for service subscription that each provider peer has to provide for managing the creation process and the data exchange between consumer and provider. Each operation thereby is implemented as a *peer method* by a peer service provider. Each peer service consumer is able to invoke these operations in order to exchange the data necessary before the actual peer service invocation can be performed. The proposed general process of a peer that has been defined in Definition 3-11 can now be adapted to model a peer adopting the provider role:

$$Supplyprocess_{Provider} = !\, port_{supply} \mid Methods$$
$$Methods = createService(sid).B_{Creation} \mid$$
$$\mid getRemotePart(sid, reply).B_{Remote}$$
$$\mid getCompAgent(sid, reply).B_{CompAgent}$$
$$\mid getHandler(sid, reply).B_{Handler}$$
$$\mid subscribe(myRecord).B_{Subscription}$$

Each of this method is accessed through the universal supply port $port_{\text{supply}}$ provided by each provider peer. If necessary, the reply port can later on be used to send back the result of the respective methods. For instance, method *getDefaultHandler* can use this port to send back the handler to the consumer. Each method is explained in more detail in the following sub sections. The operation of discovering an advertisement (step 1) is not a part of the provider. A consumer has to forward the query request to its known rendezvous peers.

A peer adopting the consumer role has to realize a deployment process within its *Supplyprocess*. This deployment process contains of a summation of the peer methods to be invoked. During deployment, the methods defined here are able to react with the methods of a *SupplyProcess* of provider. An order for invoking the respective methods is given according the communication diagram of Figure 3-11.

$$Supplyprocess_{Consumer} := ... \mid createService(sid).port_{supply}(createService, sid) \mid ...$$

3.3.10.1 Discovering a Peer Service

The first step prior to the actual deployment process is the selection of a suitable peer service. The selection of a peer service is utilized by selecting among many advertisements a consumer has found. At this stage, the user is responsible to start the discovery process and, based on a result set, to identify the most appropriate service. In this regard, the *mental proficiency* of a user is important. He has to decide whether a peer service is applicable or not. To model this proficiency accurately is certainly an ambitious if not unfeasible task. This work simply models this process of discovering and selecting a peer service in the form of the following process.

$$UserProcess_{Consumer} \mid discover(keys, reply).\overline{discoverADS}(target, reply, key)$$

In this process, a user (represented by process *Userprocess*) issues a search requests to process *Supplyprocess.* Message *keys* represent the search key. Message *reply* has a link type. This link is later on used to send back the result list to the process *Userprocess.* Within the Supplyprocess, the *discoverADS* method is invoked, which is respon-

sible to initiate start. This is done by delegating the request to one the known rendezvous peers through the interaction with their provided queryADS ports (section 3.3.8).

Again, the users themselves are responsible to reason about the correct relationship between their expected properties and the provided (given) properties of peer services. Therefore, there is actually no need for a dynamic type check between service advertisements. Hence, no subtyping rules need to be declared. The only type check necessary is to evaluate the subtype relation between the interfaces of *all* included ports. Rules for this relation have already been declared (see section 3.3.6). Runtime checker can also evaluate the implied predicates from an advertisement. For instance, a runtime condition could be the fulfillment of the membership property of a peer service. That is, a user has to be a member of all groups a provided peer service has imposed in its advertisement. A subtyping rule expressing this fact could be formulated like this:

(*Subtype Predicate Membership*)

$$\frac{T_{prd}(service_{sid},"mem") = MEM_{sid}.toSet() \subseteq T_{prd}(Userproc,"mem") = MEM_{user}.toSet()}{T_{prd}(service_{sid},"mem") \leq T_{prd}(Userproc,"mem")} \langle \phi$$

Term MEM_{sid} is a predicate that represents the implied group memberships of the peer service $service_{\text{sid}}$ obtained by applying function T_{prd}. Term MEM_{user} represents the available memberships of a given user, who is represented by process *Userproc*. The predicate of the service has a subtype relation with the predicate of the *Userproc*, if and only if the set of memberships of the service is a subset of the memberships of the user (constraint above the line). Helping function *toSet( )* is applied to turn a predicate into a set representation. The side condition is useful to guarantee correct types for both user and service process:

$$\phi = \begin{cases} T_{process}(service_{sid}) = SERVICE\_SPEC \\ T_{process}(Userproc) = USERPROCESS \end{cases}$$

Further options for supporting the matching of services based on semantic descriptions can be found in [Zaremski and Wing, 1997].

If a user has found the right service, then the *Supplyprocess* of the consumer is able to bind the supply port ($port_{\text{supply}}$) of the provider (step 1.2). The supply port is passed as a link in the corresponding peer service advertisement. After having bound the supply port, the consumer peer can proceed with the deployment process.

3.3.10.2 Creating a Peer Service

After the user has decided which service he is willing to take (step 1.1 in Figure 3-11), the service needs to be created. During creation, instances of the components belonging to a selected peer service are generated. This instantiation process is indicated by the instantiation of the respective local and interface compositions of a selected peer service. The instantiation of the single constituting components and the establishment of the port bindings are achieved by the recursive calling structure of the system port **new**. In the following, the process of service instantiation is formalized from the viewpoint of the provider:

$$Supplyprocess_{Provider} = !\left(port_{supply}(method, message, reply).\overline{method}(message, reply)\right)$$

$$Methods = createService_{onePerCust}(sid, reply).\mathbf{new}(CoI_{sid}, CoL_{sid}) \mid usedServicesProcess$$

By invoking method *createService*, an instance of peer service with the unique identifier *sid* is generated. This is in turn achieved by passing the arguments CoI_{sid} and CoL_{sid} to the system port **new**. The produced instances (see the corresponding semantics of

this system port below) are placed to a process called *usedServicesProcess* that collects all instantiated peer services (i.e. their corresponding components).

The invocation of this version of the createService method creates a new instance of both interface and local composition each time a consumer requests a service. This way, each consumer is associated with a private service (*one service per customer*). This notion of service invocation is useful to maintain a state across several port calls by the same consumer. Sharing violations with other consumers cannot occur. In certain application scenarios, it is useful to share the state among many consumers (e.g. Whiteboard application, chat tool). To support this case (*shared object*), only a new interface composition part of a dedicated peer service is generated and bound to an existing local part:

$$Methods = createService_{Shared}(sid, reply).\mathbf{new}(CoI_{sid}) \mid usedServicesProcess$$

This method assumes that a local composition part CoL_{sid} is already placed (i.e. predeployed) in the *usedServicesProcess* process. Any time a new interface composition CoI_{sid} is added to this process, it is bound with the appropriate local composition part. While the first solution may potentially lead to resource exhaustion, the latter suffers from potential sharing violations. A combination of both ways can be modeled by introducing a service pool. Here, a restricted number of predefined local composition parts serve a potentially huge number of consumers. The *service pool* variant for service creation could be formalized in the following manner:

$$Methods = createService_{Pool}(sid, reply).(fetchService(CoI_{sid}).\mathbf{add}(CoL_{sid}) \mid \mathbf{new}(CoI_{sid})) \mid usedServicesProcess$$

Method *fetchService* method returns an instantiated local composition part from a service pool. This part is added to the *usedServicesProcess* process by the system port **add**. Additionally, a new interface composition is placed to that process. A precise review of (even more) strategies for generating a service can be found in [Cervantes and Hall, 2005]. The identifiers for the service creation possibilities covered in this section are adopted by their work.

The consumer creates a service by passing the corresponding *sid* identifier to the provider. The *createService* method can be invoked by any process such as a user interface process, in which a user a selected a dedicated peer service. The interaction between such a process and the *createService* method is only adumbrated:

$$UserProcess_{Consumer} \mid createService(sid).\overline{port_{supply}}(createService, sid)$$

In this example, a user (represented by process *Userprocess*) sends a message sid to the Supplyprocess of his environment indicating which service is to be created on provider site. Then, along the supply port, the concrete request for service creation can be passed to the service provider.

During the creation of peer service within a provider peer, the provider is itself able to resolve third-party peer services from other peers in order to set up *composite peer services*. This is the case if the peer service is composed out of many different peer services (section 3.3.12). The nested creation of services is indicated by step 2.1.

3.3.10.3 Obtaining the Interface Composition Part of a Peer Service

The most important part to be exchanged during service deployment is the interface composition part *CoI* of the selected peer service. Again, *CoI* constitutes the interface for accessing the functionality of a peer service. During this phase, the integral com-

ponents of *CoI* are migrated on the target remote peer with the help of the method *setCompositionPart)*. This method must be provided by service consumers and is invoked by the providers through the passed reply port. The migration itself utilizes the *migrate* axiom to spawn a new process to a given set of (already running) running processes and, at the same, to keep a reference of the migrated service on provider side. A default implementation for the service provider could be modelled as follows:

$$Supplyprocess_{Provider} = !(port_{supply})$$

$$Methods = getRemotePart(sid, reply).\overline{reply}(CoI_{sid}).\mathbf{add}(CoI_{sid}) \mid RemoteParts \ldots.$$

$$RemoteParts = \{CoI^1_{sid}\} \mid \ldots \mid \{CoI^i_{sid}\} \mid \ldots \mid \{CoI^n_{sid}\}$$

The message *sid* denotes the request for the interface composition of the provided peer service with the unique identifier *sid*. The process migration along port *reply* replaces each migrated instance of CoI_{sid} by a remote proxy $\{CoI_{\text{sid}}\}$. This way, the remote part can be addressed and accessed remotely by the local composition part.

The consumer peer invokes the *getRemotePart* method along the supply port. This method invocation is part of the *Supplyrocess* process of each consumer. Also, a consumer has to provide the reply port *setCompositionPart* being invoked by a provider to spawn the new composition part *CoI* into its environment:

$$Supplyrocess_{Consumer} = \ldots\overline{port_{supply}}(getRemotePart, sid, setRemotePart)\ldots \mid$$

$$setRemotePart(Process).\mathbf{add}(Process) \mid RemoteParts$$

$$RemoteParts = CoI^1_{sid} \mid \ldots \mid CoI^i_{sid} \mid \ldots \mid CoI^n_{sid}$$

The system port **add** places the obtained interface composition parts to the *RemoteParts* process.

3.3.10.4 Obtaining the Composition Agent for the Interface Composition

The principle task of a composition agent is to execute *adaptation actions* triggered by a service provider or by a consumer on running instances of a local or an interface composition (see section 4.1.3 for more explanations). For each composition part, a separate agent is available. The composition agent for an interface composition is always migrated to the remote consumer peer environment. If the user of a peer service is the local operator, this agent naturally resides within the local environment. Any composition agent needs to be registered within the provider peer environment, so that it can be notified about recent adaptation actions.

The implementation of this method is analogous to the method for migrating interface compositions to a target peer. A provider implements a method called *getCompAgent* that takes a unique identifier of a service. Based on this identifier, the respective agent is returned along the provided reply port. CA_{sid} denotes an agent that administrates the corresponding peer service with the identifier *sid*:

$$Supplyprocess_{Provider} = !(port_{supply})$$

$$Methods = getCompAgent(sid, reply).\overline{reply}(CA_{sid}).\mathbf{add}(CA_{sid}) \mid RemoteAgents \ldots.$$

$$RemoteAgents = \{CA^1_{sid}\} \mid \ldots \mid \{CA^i_{sid}\} \mid \ldots \mid \{CA^n_{sid}\}$$

The process *RemoteAgents* contains proxies of all migrated composition agents. This process is later on used to address these agents, for instance, for delegating adaptation methods to all dependent remote consumer peers. A consumer peer calls the *getCompAgent* method within its *Supplyprocess* process:

$$Supplyprocess_{Consumer} = ...\overline{port_{supply}}(getCompAgent, sid, setRemoteAgent)...$$

$$setRemoteAgent(Process).\mathbf{add}(Process) \mid LocalAgents$$

$$LocalAgents = CA^1_{sid} \mid ... \mid CA^i_{sid} \mid ... \mid CA^n_{sid} \mid$$

Process *LocalAgents* collects the composition agents of all subscribed peer services. Therefore, for each obtained peer service, a local agent is available. The *SupplyProcess* process of the corresponding provider peers invokes method setRemoteAgent (passed as reply port) to pass the agent.

A provider also stores the composition agents for the local composition parts. The process of generating and storing these agents is not formalized here. The number of composition agents for a local composition part depends on the respective instantiation strategy of the surrounding peer service (see 3.3.10.2).

Again, the composition agents are responsible to adapt a local or an interface part and, thus, a peer service at runtime. Another composition agent is generated for adapting a service composition. The creation of this agent is described in section 3.3.12

3.3.10.5 Obtaining the Default Handler

Owing to the fluctuating and dynamic character of (provider) peers, consumer peers have to monitor the status of dependent provider peers. The actual monitoring process is an autonomous process and will be explained later on. The resolution of an occurred exception is carried out by exception handlers. As motivated in the state-of-the-Art section, the intention of this work is to promote the user-involvement during exception handling. Users are not only enabled to define potential handler during composition, but also to select the most suitable one after the detection of an exception. However, the definition of a handler is a task that could potentially overstrain in particular unskilled users. Therefore, service provider are accomplished to define so-called *default exception handler* that consumer could adopt. Consumer can accept these or override these by individual handlers.

$$Supplyprocess_{Provider} = !(port_{supply})$$

$$Methods = getHandler(sid, reply).\overline{reply}(HA_{sid}).\mathbf{add}(HA_{sid}) \mid RemoteHandler\ ....$$

$$RemoteHandler = \{HA^1_{sid}\} \mid ... \mid \{HA^i_{sid}\} \mid ... \mid \{HA^n_{sid}\}$$

HA_{sid} is the default handler being pre-defined for a peer service with the identifier *sid*. Like agents and interface compositions, all transferred remote handlers are stored as proxies in process *RemoteHandler*. The consumer part is formalized as follows:

$$Supplyprocess_{Consumer} = ...\overline{port_{supply}}(getHandler, sid, setHandler)...$$

$$setHandler(Process).\mathbf{add}(Process) \mid DefaultHandlers$$

$$DefaultHandlers = HA^1_{sid} \mid ... \mid HA^i_{sid} \mid ... \mid HA^n_{sid}$$

3.3.10.6 Subscription of a Consumer Peer for a Peer Service

The information flow mentioned so far originates from a service provider towards its consumer. However, an information flow from consumer towards provider is also conceived during the so-called consumer subscription process. This is the case during the process of consumer subscription. This process is necessary especially for peer providers, so that they can be aware about peer consumers using one of their provided peer services. This way, consumer dependencies can be traced. Tracing this type of dependency is of major relevance during the adaptation of peer services. If an adapted

peer service holds a dependency on other consumer peers, then the adaptation attempt must be carried out carefully. The unrevealing of existing consumer dependencies is therefore a major requirement. By doing so, each dependent consumer peer can be notified about forthcoming adaptation events. The *analysis of consumer dependencies* is discussed at length in section 4.1.2. Information about dependent consumer peers is also essential during exception handling. If an exception has been detected, then all dependent consumer peers could also be notified as well. This process of forwarding of exception is termed *exception cascading* (see section 4.3.3).

To achieve the consultation between provider and their consumers for exception cascading and for the analysis of consumer dependencies (section 4.1.2), consumers register to any provider peer from which they consume a service. In this architectural style, the consumers' peer addresses *pid* and the unique identifiers *sid* of the dependent peer services are crucial informations that need to be passed. Also, a dependency value *depValue* indicating the relevance to a dependent value is to be passed. A consumer can later on update this dependency value. All necessary subscription data are passed within a single message $consumerRecord_{\text{sid}}$, where *sid* is the unique identifier of the provided peer service. The provider site is formalized in the following way:

$$Supplyprocess_{Provider} = !(port_{supply})$$

$$Methods = subscribe(consumerRecord_{sid}).\mathbf{add}(consumerRecord_{sid}) \mid SubscribedPeers$$

$$SubscribedPeers = (consumerRecord^{1}_{sid} \mid ... \mid consumerRecord^{n}_{sid})$$

Process *SubscribedPeers* contains all peers that hold dependencies to (at least) one o the provided services. The consumer view is formalized in this manner:

$$Supplyprocess_{Consumer} = ...subscribe(myRecord_{sid}).\overline{port_{supply}}(subscribe, myRecord_{sid})... \mid$$

$$addRecord(myRecord).\mathbf{add}(myRecord) \mid DependentProviderPeers$$

$$DependentProviderPeers = (myRecord^{1}_{sid} \mid .... \mid myRecord^{n}_{sid})$$

The consumer also stores the subscription record by invoking port *addRecord*.

3.3.11 The Invocation of a Peer Service

The direct invocation of a peer service, that is, the local invocation of a port, which is part of the peer service's public interface (API), can be modelled in the same manner as the interaction between two component ports within a local composition (section 3.3.6). A critical part is the process of delegating a service execution from the interface composition *CoI* to the local service composition *CoL* along their façade ports. Interaction between façade ports is also indicated as *remote interaction* (Figure 3-12).

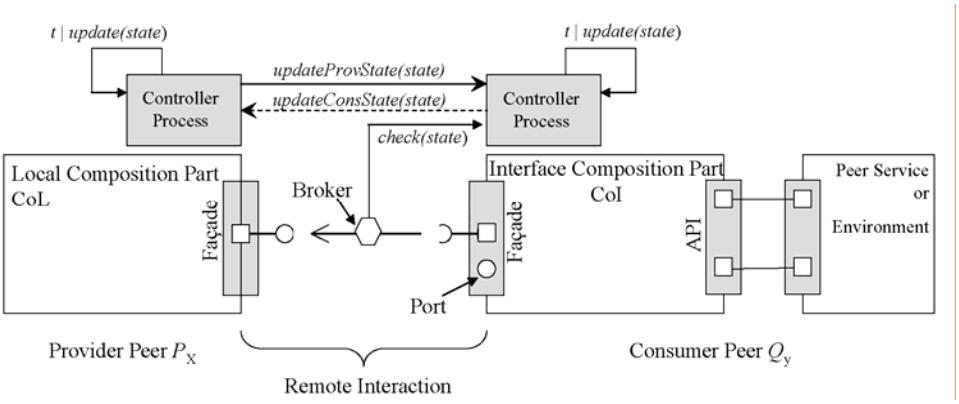


Figure 3-12: Port communication between the facades (remote interaction)

A remote interaction between the local and the interface composition becomes a problem, when the process hosting the local composition *CoL* (provider peer P_x) becomes *unavailable*. An exception occurs, when a required port from an interface composition is willing to interact with a provided port from a local composition (see constellation in Figure 3-12). This in turn results in unpredictable process behaviour within all peer services that depend on the Interface Composition *CoI*.

In order to guarantee port interaction in the presence of peer (process) failures, each consumer peer has to maintain the current state of their dependent provider peers, from which they have deployed a peer service. The maintenance of peer states is covered by process *Controllerprocess*, located on both consumer and provider site (Figure 3-12). The structure of such a controller process from the perspective of a provider can be modelled as follows:

$$Supplyprocess_{Provider} = !(port_{supply})$$

$$!updateProvStatus(status_{own}, reply).\overline{port_{supply}}(status_{own}, updateConsStatus)$$

$$!updateConsStatus(status, reply).\mathbf{add}(status) \mid Controllerprocess$$

$$Controllerprocess = status^{1}_{sid} \mid ... \mid status^{n}_{sid}$$

Any system process can invoke port method *updateProvStatus* in order to convey the current own *status* (as a message) to all recently subscribed consumers along their supply ports (it is assumed that the provider maintains somewhere the linked supply ports). As a reaction of the status update of a provider, a consumer can itself send back its own status along reply link. This process for a consumer is as follows:

$$Supplyprocess_{Consumer} = !(port_{supply})$$

$$!updateProvStatus(status, reply).\overline{reply}(status_{own}).\mathbf{add}(status)$$

$$... \mid Controllerprocess$$

$$Controllerprocess = status^{1}_{sid} \mid ... \mid status^{n}_{sid}$$

The status information about dependent consumers can be useful for a provider peer for the purpose of checking predefined integrity constraints that encompass conditions about dependent consumers (see section 4.2 for more details). Method *updateProvStatus* is invoked (along the supply port) by each provider peer in regular intervals. The following types of states are provided:

$$status = [active, failed, unavailable, inconsistent]$$

Status *active* denotes a running peer. At any time, a provider peer can send status *unavailability* to announce the immediate unavailability of it. During an *inconsistent* state, a peer is running, but cannot guarantee a proper run of the peer service. The reason for an inconsistent state might be the fact that the provider peer itself has detected the failure of a dependent peer on which the functionality of the provided peer service is depending. In addition, a consumer peer is able to actively update the status of a provider to status *failed*, if for a predefined time no update signals have arrived. During the update method, the previous state must be deleted.

The status of a distinct peer service PS_{sid} is derived by the status of the peer providing this service and must be encoded in the form of a special port and placed within the *Controllerprocess*, so that an external process can request the current status of that peer service (e.g. peer-sid-active(message), peer-sid-inconsistent(message)). This external process is termed *broker*. A broker has the responsibility to check the status of a peer, before an interaction between two façade ports can be pursued. This way, each

binding between two façade ports $port_i$ and $port_j$ is assigned a broker process $Broker_j^i$. A broker process is not directly modeled but inserted in the condition part of the reduction rule defining the operational semantics for remote interaction:

$$\frac{\overline{PS_{sid}^{pid}\,active}(message).Broker_j^i \longrightarrow Broker_j^i \quad PS_{sid}^{pid}\,active(message).Controllerprocess \longrightarrow Controllerprocess}{\left(\overline{port_{interaction}^i}(out, portA_{reply}).A \mid port_{interaction}^j(in, portB_{reply}).B\right)} \langle \phi \rightarrow \left(A \mid [out/in, portA_{reply}/portB_{reply}]B\right)$$

The following side condition signifies that the ports must be part of the facades *CoI* and *CoL* belonging to peer service PS_{sid} and that PS_{sid} must be a consumed remote peer service (expression F_{CoI}^{sid} denotes the façade of the interface composition of peer service with the unique identifier *sid*):

$$\phi = \begin{cases} T_{port}(\overline{port_{interaction}^i}) \le T_{port}(port_{interaction}^j) \\ port_{interaction}^i \in F_{CoI}^{sid} \wedge port_{interaction}^j \in F_{CoL}^{sid} \\ T_{process}(A) = COMPOSITION, A \text{ is an interface composition} \\ T_{process}(B) = COMPOSITION, B \text{ is a local composition} \end{cases}$$

In this reduction rule, the broker process is able to react with the prefix $PS_{sid}\,active(message)$ of the *Controllerprocess* expressing the availability of the remote provider peer. If this condition is met, the interaction can be carried out without risk. For the case that a provider is unavailable, the interaction cannot be carried and is halted. Instead, adequate exception handler routines must be selected and invoked. This case is illustrated in section 4.3.

3.3.12 Composition of Peer Services

Peer services stemming from different provider peers can be composed with local peer services to a more complex *service composition*. A service composition can function as a local application that is only used internally within a single peer environment. According to this composition type, the operator of that peer environment acts as an assembler. A service composition may also define a new peer service that can be published, discovered and integrated third-party consumer peers. This type of peer service is referred as a *composite peer service*. Here, the assembler not only acts as the assembler of that composite peer service but also as the service provider of it.

This section defines a service composition in a generic way. As a restriction, only bindings between ports being part of the public interfaces *API* (part of the interface composition *CoI*) of (remote and local) peer services can be established. Besides, only peer services, that is, no distributed compositions can be taken for a composition. A service composition can now be defined in a formal way as follows (Definition 3-17):

Definition 3-17 (Service Composition). Let $PS = \{P_1,\ldots,P_n\}$ be a set of public peer services, $B = \{(P_i.port, P_j.port_j)\}$a set of port bindings with $P_i.port_i \in API_{P_i}$. Then the tuple $SC = (PS, B)$ is a service composition.

□

Definition 3-18 (Process of a Service Composition). The process of a service composition *SC* = (*PS*, *DCs*, *B*) is arranged as follows:

$$ServiceCompositionProcess = PeerServices \,|\, Bindings \,|\, Reflectionprocess$$

$$PeerServices = InterfaceParts \,|\, LocalParts$$

$$InterfaceParts = CoI^{1}_{sid} \,|\, ... \,|\, CoI^{i}_{sid} \,|\, ... \,|\, CoI^{n}_{sid}$$

$$LocalParts = CoL^{1}_{sid} \,|\, ... \,|\, CoL^{i}_{sid} \,|\, ... \,|\, CoL^{n}_{sid}$$

$$Bindings = !(\prod_{i=1}^{n} binding_i(API_{sid}.port_k^{contract}, API_{sid}.port_j^{contract}))$$

The process *Reflectionprocess* can be used to alter the bindings between API ports or to add and to delete services from a composition:

$$\begin{aligned} Reflectionprocess = {} & !addService(peerService).\mathbf{add}(peerService) \\ & |!deleteService(peerService).\mathbf{delete}(peerService) \\ & |!addBinding(binding).\mathbf{add}(binding) \\ & |!deleteBinding(binding).\mathbf{delete}(binding) \end{aligned}$$

The process type of a service composition process depends on the viewpoint. The template of a service composition has the process type:

$$T_{\text{process}}(\ ServiceComposProcessTemp\ ) = \text{SERVICECOMP\_SPEC}$$

The instance of a peer service has the process type:

$$T_{\text{process}}(\ ServiceComposProcessInst\ ) = \text{SERVICECOMPOSITION}$$

□

A service composition can be published again as a peer service, which is called a composite peer service. Like a conventional peer service (section 3.3.7), a composite peer service must also hold an advertisement and an API as shown in the next definition:

Definition 3-19 (Composite Peer Service). Let *SC* be a service composition and $API_{SC}(A, ads)$ the public interface of that composition including an advertisement *ads*. Let *sid* be a universally unique integer, $T_{\text{dataType}}(\ sid\ )$ = INTEGER. Then the tuple $CS = (SC, API_{SC}, sid)$ is a composite peer service**.**

□

The process of a composite peer service conforms to the service of a service composition together with an additional parallel process *APIprocess* for offering the public interface and the advertisement (omitted here). The template of a composite service has the process type:

$$T_{\text{process}}(\ CompositeServiceProcessTempl_{sid}\ ) = \text{COMPSERVICE\_SPEC}$$

The instance of a composite peer service has the process type:

$$T_{\text{process}}(CompositeServiceProcessInst_{sid}\ ) = \text{COMPSERVICE}$$

3.3.13 Execution Models

Depending on the distribution of bindings between API ports, different *execution models* are possible for executing a service composition and a composite peer service, respectively. If bindings only apply within interface composition parts, all bindings between them need to be established in the consumer environment. Here, the provider

environment is only responsible to provide the service but not to execute it. This execution model is termed *basic execution model* (Figure 3-13).

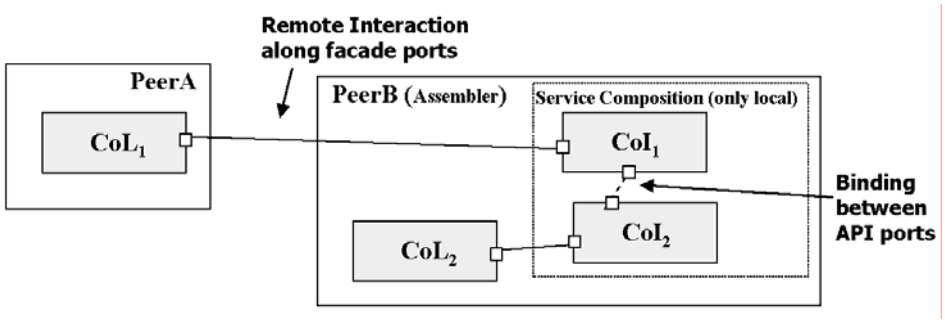


Figure 3-13: Basic execution model of a service composition

Here, the service composition realizes a local application that is only accessible for PeerB (the assembler). It consists of a local peer service (i.e. the local and the interface part reside at the local environment) and a remote peer service of PeerA.

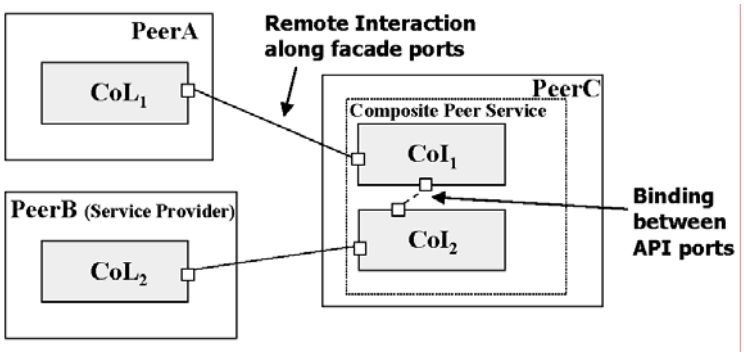


Figure 3-14: Basic execution model for composite peer services

PeerB is also able to publish the service composition as a composite peer service, so that other peers can locate and bind it (see Figure 3-14). In this example, PeerC acts as the consumer of the composite peer service that has been composed and eventually published by PeerB. To realize this model, the peer service advertisement has to include the bindings between API ports as a semantic property field (see Definition 3-8). This way, the local peer environment of PeerC is capable of identifying the necessary bindings between API ports.

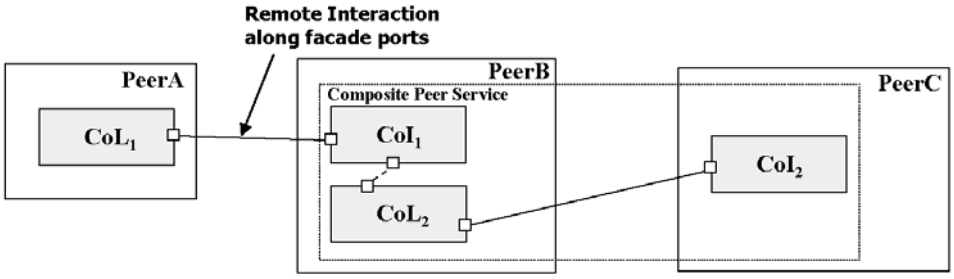


Figure 3-15: Distributed execution model for composite services

In the case that an external peer service is bound with a local composition part of a peer service, the service provider is responsible for establishing the local binding and to execute the respective local and interface parts of the composite peer service (Figure 3-15). During the creation of the peer service (step 2 in Figure 3-9), a provider peer can be requested to integrate dependent peer services and bind them on behalf of

the requesting consumer peer (step 2.1 in Figure 3-9). This execution model is called the *distributed execution model*. More information on these execution models can be found in the reference implementation of $SO_{P2P}A$ (section 6.5).

The above definition of service composition does incorporate neither exception handlers nor composition agents into a given composition. These elements can be integrated by the component assembler if desired. Note that each peer service composition can also run without any exception handlers as well.

Composition Agents

If a service composition is deployed within a peer environment, a composition agent is automatically generated in it as well. This agent is responsible for delegating adaptation actions to a running instance of a service composition. With such an agent, it is possible, for instance, to add or to delete services to a service composition at runtime (see section 4.1.3). The composition agent always resides in the consumer environment, that is, the environment in which the execution of a composition has been initiated. It is also registered in the provider peer. Whenever a provider adapts the template of a service composition at runtime, the remote agents can be addressed in order to delegate the effected adaptation actions accordingly (see more information in 4.1.3).

3.4 Summary

This chapter has presented the fundamental aspects of the $SO_{P2P}A$ architectural style. So far, mainly aspects on *deployable processes* (components, peer services, binding components and services) as well as on *environment processes* (i.e. peer process, deployment aspects) have been specified. The next chapter focuses on aspects covering the aspired *diagnostic concepts* (e.g. exception detection, consumer analysis) and *manipulation concepts* (e.g. adaptation of process structures).

Chapter 4
Component-based Adaptation Methods in $SO_{P2P}A$

This chapter presents formalisms for adapting, that is, for manipulating process structures that form a service-oriented peer-to-peer architecture according to the $SO_{P2P}A$ architectural style. At first glance, the presented adaptation methods are targeted for end-users, enabling them to tailor process structures, for instance, as a reaction on changed requirements (section 4.1.3 and 4.1.4). In addition, the style entails to use the same adaptation methods to handle exceptions that could occur due to the failure of single peers (section 4.3). In order to ensure the correct adaptation of a service without violating consumer dependencies, a provider is asked to analyse consumer dependencies and to follow the procedures of an *adaptation policy* (section 4.1.2).

Another important concept also introduced in this chapter is the notion of an *integrity constraint* (section 4.2). Integrity constraints aim at enhancing the reliability of peers and peer services that intend to collaborate in a peer-to-peer architecture. An integrity constraint conforms to a *contract* negotiated between a provider and consumer of a service mainly regulating the presence of a service in a given context. The unavailability of a service may violate such contracts, which can be handled by means of the aspired adaptation methods as well. All relevant concepts concerning the adaptation of process structures are treated in a separate section.

4.1 The Adaptation of a Peer Service

Both provider and consumer of a peer service should be accomplished to adapt a peer service as new requirements arrive or exceptions occur in the context of a service. The offered adaptation methods to master these adaptations should comply with *component-based adaptation methods* obeying the requirements elicitated in section 2.5.4.2. Adaptation methods should be applicable on various levels of granularity, that is, on ports, components, services and service compositions. Based on these elements, concrete methods are formulated in terms of deleting or adding single ports of components or services, changing bindings between components, as well as deleting or adding components or services from compositions *at runtime*. Adaptation routines are defined in an abstract way, so that both an end-user and a system agent are capable of adapting a composition. End-user *tailoring* is the focus in this section. Semi-adaptive adaptation methods triggered by system agents can be used for adapting compositions in the case of a failure, which are covered in section 4.3.2.

A single adaptation step (e.g. deleting a port or changing the connection between two components) is regarded as an *adaptation action*. A set of coherent adaptation actions can be aggregated as an *adaptation bundle*. An adaptation bundle can only be executed completely or not at all. Adaptation actions and bundles should be carried out on process templates (e.g. a component, or a service composition template) and on running instances of them. Adaptations to process templates correspond to long-term changes in order to the structure of a process that will automatically have an effect on all future instances as well as all currently running instances. These need to be stored persistently. Adaptations to running process instances have an impact only on the running instance but not on future instances of the process template. The latter adaptation variant is the target for end-users, for instance, to hide functionality of a service.

Although both providers and consumers possess the ability to adapt services and compositions, both have different *adaptation rights* to adapt them. Each adaptation right exposes different process behaviour, say, for tailoring services from a consumer perspective and from a provider perspective, respectively. Each special process behaviour uses some general behaviour that implements the underlying adaptation mechanisms. The overall process or *functional behaviour* for adapting a composition can be illustrated by a use case diagram (Figure 4-1). This diagram represents the necessary processes as use cases (oval shapes) that interact with two different actors Consumer and Provider. Each use case consists of a name and of a process identifier. The identifier represents the process realizing the dedicated use case behaviour later on in the formal, algebraic notation. The two usage perspectives can now be elaborated:

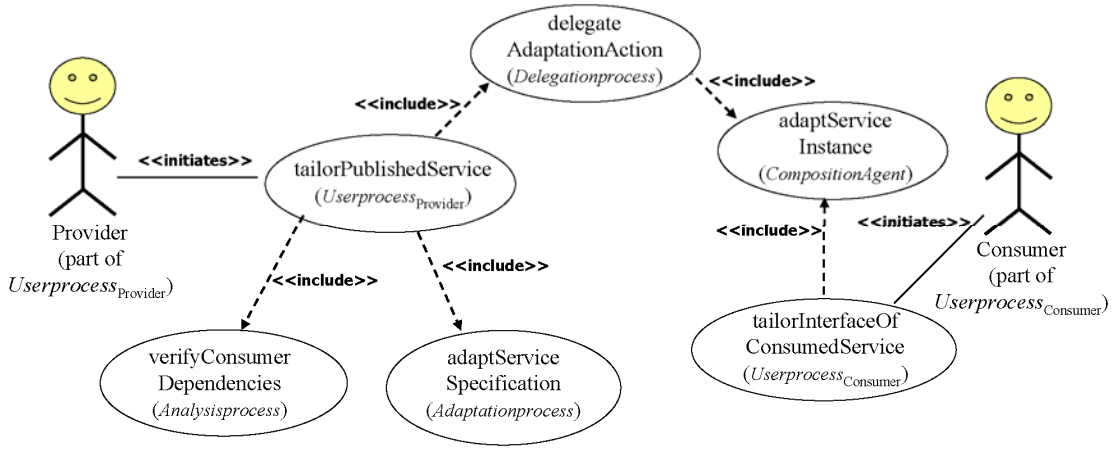


Figure 4-1: Overview of the desired functionality for service adaptation (UML use case diagram)

- A provider should be able to adapt a published peer service (base use case tailorPublishedService). This can be either a single service or a service composition that has been published as a composite peer service. Each adaptation step affects the local template specification of a peer service at first (use case adaptServiceSpecification). Afterwards, these steps are delegated to all running *instances* of that peer services being deployed in the local environment (i.e. local composition part) and on consumer peer environment (i.e. interface composition part). This behaviour is illustrated by the included supplier use cases delegateAdaptationAction and adaptServiceInstance. In order to avoid the violation of consumer dependencies by the execution of an adaptation bundle, all dependent consumers must be analyzed according to the respective adaptation policies implied by the involved peer groups (use case verifyConsumerDependencies). Apparently, the same methods

can be used to adapt a local service composition that is not published as a composite service. Here, no analysis is necessary.

- A consumer should be able to adapt the interface composition part *CoI* of a consumed peer service (base use case tailorInterfaceOfConsumedService). Such an adaptation aims at varying the local interface of a peer service only. Any adaptation action only affects the local instance but has neither effect on other instances deployed on third-party peers nor on the actual service definition residing at the provider's site. For the actual adaptation, supplier use case adaptServiceInstance is used. If necessary, adaptations might be made persistently so that they can be presumed after the service is re-instantiated. If the consumed peer service is a composite service, the consumer is also capable of tailoring the service composition process (e.g. by deleting a service and adding a new one).

The base use cases for tailoring process templates is adaptServiceSpecification. In $SO_{P2P}A$ process *Adaptationprocess* realizes this use case. In this process, all relevant component-based adaptation methods are formalized for adapting process templates. This process is described in the next section.

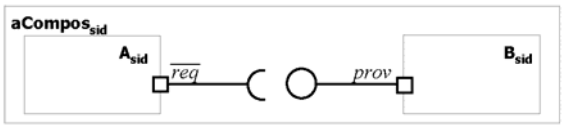
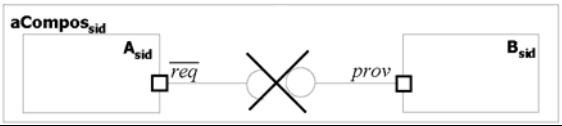
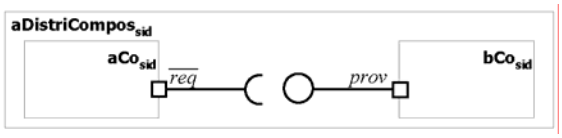
4.1.1 Component-based Adaptation Methods (*Adaptationprocess*)

Component-based adaptation methods refer to the same methods as for constructing component-based applications. In accordance to $SO_{P2P}A$, the same methods conceived for setting up process structures can now be applied for adapting them. Adaptation can thereby be pursued on both template and instance of processes. In the algebraic notation, adaptation methods are provided by a process called *Adaptationprocess* which is attached to the *Supplyprocess* of a peer. The general structure of this process is:

$$Supplyprocess = !\, port_{supply} \mid Adaptationprocess$$

$$Adaptationsprocess = !adaptMethod(params).B_{adaptMethod} \ldots \mid \ldots$$

Although other adaptation routines are conceivable (e.g. the renaming of ports), the focus will be set on the following routines as outlined in Table 4-1. Each adaptation method is formalized as a separate method together with additional conditions.

Adaptation Method	**Illustration**
addBinding(aCompos$_{\text{sid}}$*,* A_{sid} *.req), B_{sid}.prov)* Condition: $T_{port}(A_{sid}.req) \leq T_{\text{port}}(B_{sid}.prov)$	
deleteBinding(aCompos$_{\text{sid}}$*,* A_{sid} *.req, B_{sid}.prov)*	
addBindingFacades(aDistriCompos$_{\text{sid}}$*, aCo_{sid} .req, bCo_{sid}.prov)* $T_{port}(aCo_{sid}.req) \leq T_{\text{port}}(bCo_{sid}.prov)$ Condition: req and prov are façade ports	

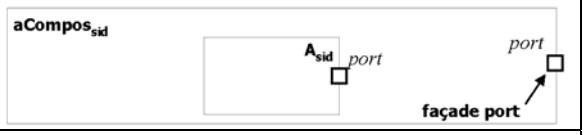
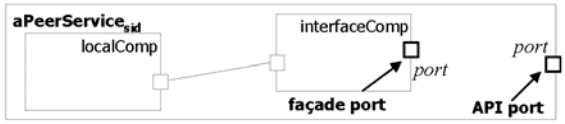
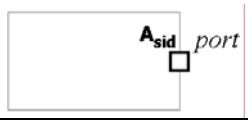
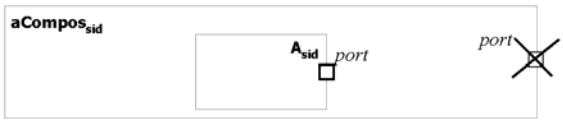
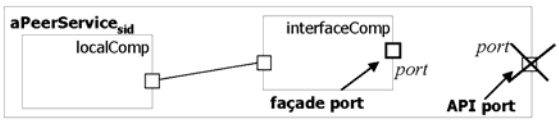
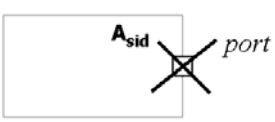
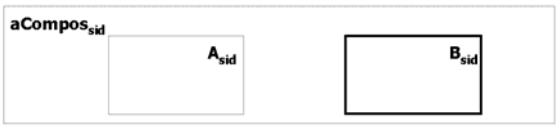
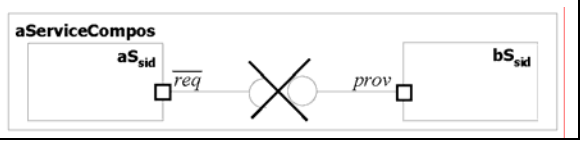
*deleteBindingFacades (aDistriCompos*sid*, aCo*sid*.req, bCo*sid*.prov)* Condition: req and prov are façade ports	
*addPortToFacde(aCompos*sid*, A*sid*.port)* Condition: *F*sid *.port = A*sid *.port*)	
*addPortToAPI(aPeerService*sid*, F*sid *.port)* Condition: *F*sid *.port = API*sid *.port*)	
*addPortToComponent(A*sid*, port)*	
*deletePortFromFacde(aCompos*sid*, F*sid*.port)* Implication*: A*sid*.port* is not deleted)	
*deletePortFromAPI(aPeerService*sid*, API*sid*.port)* Implication*: F*sid*.port* is not deleted)	
*deletePortFromComponent(A*sid*, A*sid*.port)*	
*addComponent(aCompos*sid*, B*sid *)*	
*deleteComponent(aCompos*sid*, B*sid *)*	
*addPeerService(aServiceCompos, aService*sid *)*	
*deletePeerService(aServiceCompos, aService*sid *)*	
addBindingAPI(aServiceCompos, aS*sid .req, bS*sid*.prov)* Condition: *T*port*(aS*sid *.req) ≤ T*port *(bS*sid *.prov*)	
*deleteBindingAPI(aServiceCompos*sid*, aS*sid *.req, bS*sid*.req)*	

Table 4-1: Overview on the adaptation methods available in SOP2PA

The processes $aPeerService_{sid}$, $aCompos_{sid}$, and A_{sid} represent peer services, components, and compositions from a peer service with the unique identifier *sid*. Process $aPeerService_{sid}$ is only used for altering the API of a public service. For altering the façade ports of a peer service, a distributed composition is passed as a reference that enables for accessing the façade port processes. If a port is deleted from a component, it will also be deleted from any façade or public interface. In addition, if a component is deleted from a composition, all pertaining ports are deleted from the facades and the public interface of the respective peer service.

Process *aServiceCompos* represents a service composition. All adaptation methods affect a service in terms of adding or deleting a peer service or altering the bindings between API ports. All methods are valid on both local service composition and a composite peer service. These methods for adapting a service composition play an important role for handling exceptions upon the failure of a peer process (section 4.3).

All adaptation methods are applied only on the template of a peer service, composition, or component. The following side condition applies to the *Adaptationprocess*:

$$\phi \equiv \begin{cases} T_{Process}(A_{sid}) = T_{Process}(B_{sid}) = COMPONENT\_SPEC \\ T_{Process}(aCompos_{sid}, aDistriCompos_{sid}, aCo_{sid}, bCo_{sid}) = COMPOSITION\_SPEC \\ T_{Process}(aPeerService_{sid}) = SERVICE\_SPEC \\ T_{Process}(aS_{sid}) = T_{Process}(bS_{sid}) = SERVICE\_SPEC \\ T_{Process}(aServiceCompos) = SERVICECOMP\_SPEC \vee \\ \quad = COMPSERVICE\_SPEC(\textit{for composite service}) \\ T_{Process}(port) = PORT\_SPEC \end{cases}$$

Auxiliary Adaptation Methods

Apart from the proposed component-based adaptation methods, a couple of further, rather auxiliary but important adaptation methods are necessary (Table 4-2).

Adaptation Method	**Illustration**
discoverPeerService(description,port)	Searches for a new peer service with the given description.
applyMembership	Applies for a group membership
subscribeToService	Subscribes to a service on a provider peer

Table 4-2: Auxiliary adaptation methods

Before adding a new service, a peer user first has to locate a new service within a given peer-to-peer architecture. Method *discoverPeerService* makes use of a rendezvous peer to start querying to peer service advertisements (see section 3.3.8.2). Given that the new peer requires the membership of a peer group in which the operator is not involved, adequate tools can offer a method *applyMembership* for directly applying to a membership. Finally, a user is able to subscribe to a peer service on the respective provider peer (method *subscribeToService*). In some way, these methods represent pre-conditions that need to be satisfied before the actual component-based adaptation of Table 4-1 can be applied. An accurate semantic of these methods is not given here. More information can be found in the reference implementation (DEEVOLVE) that provides suitable implementations of these methods (see 8.4.2).

Adapting Template Process Structures through the Reflection Processes

Each method embraces its individual block B_{id} to handle an adaptation request. To carry out the adaptation actions, these blocks make use of the methods defined in the *Reflectionprocesses* provided by the corresponding compositions, components, and service process templates. In the following, the blocks of three methods are illustrated for demonstrating their semantics. Method *deleteBinding* can be realized in this way:

$$deleteBinding(composition_{sid}, port_x, port_y).\left(\overline{deleteBinding}(port_x, port_y).composition_{sid}\right)$$

This method makes use of the operation *deleteBinding*, which is part of the *Reflectionprocess* of a composition specification (Definition 3-4). Adding a port to a façade is defined as follows:

$$addPortToFacade(aCompos_{sid}, port_x,).\left(\overline{addPort}(port_x).facadeProcess_{aCompossid}\right)$$

Here, the façade process of the composition *aCompos* is directly used to add a further façade port. Adding a service to a service composition can be formalized in this way:

$$addPeerService(aServiceCompos, service,).\left(\overline{addPort}(service).aServiceCompos\right)$$

In principle, all adaptation methods can be realized just by the add and delete methods defined by the reflection processes of the components, services and compositions.

All adaptation actions are invoked by the separate processes $Userprocess_{\text{Provider}}$ (see Figure 4-1). Process $Userprocess_{\text{Consumer}}$ also makes use of a similar set of adaptation methods that are implemented by process *CompositionAgent*. These methods, in contrast to the method of process *Adaptationprocess*, operate on instances of processes only. The purpose of both user processes is treated in sections 4.1.3 and 4.1.4. Before an adaptation step can be put into effect, an actor is asked to pursue an evaluation for analyzing potential consumer dependencies. This process of consumer dependency analysis is described in the forthcoming sub section.

4.1.2 Consumer Dependency Analysis

Process Analysisprocess

Before the adaptation of a published peer service can be utilized, the provider is asked to analyse potential dependencies to consuming peers that have located and deployed an instance of that peer service into their environment (process *Analysisprocess* in Figure 4-1). If consumer dependencies are available, so-called adaptation policies should be applied that dictate rules how to cope with existing dependencies. These policies are determined for each peer group and can be obtained by the corresponding group advertisement (see section 3.3.9). This way, the architectural style provides means for dealing with uncontrolled or unwary adaptations that could lead to the violation of functional dependencies in consumer peers.

The adaptation policy comes with a process *Policycheckerprocess* for executing the local analysis of consumer dependencies. This analysis process is based only on the consumer data that has been stored during the subscription process of a consumer (see section 3.3.10). That is, no further data needs to be acquired during that process, it can be carried out offline. The *Policycheckerproc*ess process can be obtained from one of the group founders and is stored within the peer environment (not further formalized).

The process of analyzing consumer dependencies can be started by invoking method *analyzeDeps* of process *Analysisprocess*:

$$Analysisprocess_{sid} =$$
$$!analyzeDeps(PolicyCheckerprocess_{pgid(sid)}, policy_{pgid}, UserRecords_{sid}).B_{analyze}$$
$$|!getReport(returnPort).B_{getReport}$$
$$|!executeStrategy(adaptationstrategy, UserRecords_{sid}, data)$$

This method takes the corresponding $PolicyCheckerprocess_{\text{pgid(sid)}}$ entailed by the peer group with the identifier *pgid*. The suffix *sid* indicates the peer service to be adapted, function *pgid(sid)* delivers the identifier of the group to which the service is associated. Message $policy_{\text{pgid(sid)}}$ represents the recent adaptation policy. Both the policy and the checker process are separated, because the policy itself can change during the lifetime of a peer group. Message $UserRecords_{\text{sid}}$ denotes all user subscription data of consumers that have subscribed for peer service with the identifier *sid.* Within the (unspecified) block B_{analyze}, the actual analysis process is carried out. The process *Analysisprocess* and its methods run in parallel to processes $Userprocess_{\text{Consumer}}$ and $Userprocess_{\text{Provider}}$, respectively. It is shown later that both user processes must consult the *Analysisprocess* before processing with actual service adaptation.

The result of the analysis is a *report*. The report consists of a justification to what extent potential consumer dependencies might influence an immediate adaptation. Depending on the number of dependencies or depending to any weighted, summarized, or any other analytic presentation of consumer dependencies, certain so-called *adaptation strategies* might be suggested. A possible strategy would be, for instance, to notify all dependent consumers, which are members of the same peer group about the forthcoming adaptation request. Note that the analysis and the proposal of an adaptation strategy incorporate only user subscription data of consumers who are members of the given peer group providing the adaptation policy $policy_{\text{pgid}}$. This confinement reduces the (potentially huge) number of consumers to be considered during the analysis process. One of the user processes can obtain the report by invoking method *getReport*. It is sent back along the link port *returnPort*.

A single strategy can be invoked by calling method *executeStrategy*. This method needs three messages as input. First, the chosen adaptation strategy is provided (message *adaptationStrategy*). Secondly, message $UserRecords_{\text{sid}}$ represents all consumers that are combined with adaptation strategy. For consumer notification, for instance, this message includes all consumers that need to be notified before an adaptation can be proceeded. As a third argument, message *data* represents any kind of additional information needed for executing the strategy. Regarding the notification option, this could be a text explaining the purpose of the planned adaptation. The concrete formalization of adaptation strategies is not provided. Also, the style makes no assumptions on how many adaptation strategies should be given. The notification strategy, for example, could make use of the message exchange capabilities of a peer environment (see section 3.3.8.3). More information on adaptation policies and strategies is found in the reference implementation (chapter 7).

At all times, any user process can obtain an updated report that respects the possible impacts of the adaptation strategies. Thus, a final report would state that all dependencies have been considered and treated with respect to the policies. Then and only then, an adaptation can be initiated.

Initiating the Adaptation

The prerequisite before an actor is able to initiate an adaptation, is that all consumer dependencies have been treated with respect to the adaptation policies of the peer

group the service associates. This assumption includes that, if demanded, appropriate adaptation strategies have been applied for establishing a state that is desired by all adaptation policies (e.g. all dependent consumers have been notified). This state can only be checked by verifying the report of an analysis. If an actor (represented either by process $Useprocess_{\text{Consumer}}$ or $Useprocess_{\text{Provider}}$) is willing to initiate an adaptation (method *initiateAdaptation* of process *Adaptationprocess*), he has to provide the current report as an argument:

$$Userprocess_{\substack{Consumer \\ Provider}} \mid \underbrace{initiateAdaptationSession(report).B_{startSession}}_{Adaptationprocess} \langle \phi$$

Only if the report gives an indication that all requirements of the adaptation policies have been satisfied, the adaptation session can be initiated. This condition is provided (in a semi-formal way) as a side condition:

$$\phi \equiv \begin{cases} all\ dependencies\ have\ been\ treated\ w.r.t.\ to\ the\ associated\ policies \\ all\ implied\ adaptation\ strategies\ have\ been\ applied \end{cases}$$

After the adaptation session has been started (indicated by process $B_{\text{startSession}}$ but not further formalized), all other adaptation steps can be invoked (see section 4.1.3).

Formulating Reputation Values

The restriction, that an adaptation can only be started after an analysis has been carried out thoroughly, is certainly a strict condition. One could weaken this condition by inducting the process of consumer analysis as a *voluntary* process a provider *might* conduct. Having such process, however, one has to introduce a way for *avenging* the careless or defective adaptation of a published peer service without regarding any consumer dependencies. The higher the number of such bad adaptations, the less becomes his overall reputation within a community (e.g. in this peer groups). In order to tackle this problem, the proposed style recommends the notion of *reputation* as an important indicator for discriminating between confidential and untrustworthy service providers. Any consumer is capable of formulating reputation values for service providers. High reputation values are assumed for confidential providers, low values for untrustworthy providers. Again, the reputation is mainly influenced by the way of how a provider pursues the adaptation of published peer services. The style makes no statement concerning concrete values (i.e. a minimum or maximum value).

An automatic derivation of reputation values is certainly complex and is not covered in this work. Instead, human users conceive reputation values by mental processes based, for instance, on the awareness capabilities of them. By means of the user process *Reputationprocess,* a user is able to define reputation values. It is assumed that this process provides an adequate interface and support for entering such values. A reputation value is represented by a *reputation advertisement*.

The *Reputationprocess* makes use of the routing capabilities of a peer in order to publish and to propagate a reputation advertisement. The interaction is not further specified. Any consumer can then refer to such advertisements. They can influence the decision whether a peer service from a distinct provider can be trusted for a local service composition (see section 3.3.7 and definition of a service advertisement).

4.1.3 Tailoring a published Peer Service

This process is an abstraction that encapsulates any kind of behavior for adapting (or tailoring) a published service. The process contains an interface to an external actor

(Provider), which is – most typically – a human user. The provider is able to use the interface for controlling the tailoring process. As part of the *Userprocess* of a peer environment, the actual interface from this process to the respective actor is not further specified. The main idea of *Userprocess* is to pass all relevant input messages that are necessary to pursue an adaptation action (e.g. the ports, component process). The process $Userprocess_{\text{Provider}}$ runs in parallel to the process *Adaptationprocess*.

$$Userprocess_{Provider} \mid \underbrace{(!adaptationMethod_1.B_1 \mid ... \mid !adaptationMethod_n.B_n)}_{Adaptationprocess}$$

Port $adaptationMethod_i$ represents the *i*-th adaptation method of an *Adaptationprocess*. The process $Userprocess_{\text{Provider}}$ is responsible to query (or to ask the user for) the necessary input values necessary for an adaptation. This data is prepared and eventually passed to the *Adaptationprocess*. The interaction between both processes is not further formalized. Adaptations bundles are created and maintained internally in $Userprocess_{\text{Provider}}$ process. This process ensures that a bundle is completely executed. The *Adaptationprocess* only receives single adaptation requests. The maintenance of adaptation bundles is not further specified.

CompositionAgent for Maintaining Local and Remote Compositions

In order to forward the changes made to a distinct component specification to all its running instances, the respective *CompositionAgent* processes must be consulted. Recall that for each instance of a local composition part *CoL* and for each interface composition part *CoI* (deployed at remote peers), a single *CompositionAgent* process has been generated and deployed. If a service composition or composite peer service is deployed, a third *CompositionAgent* process is available. Each *CompositionAgent* provides two ways for adapting process instances. First, an external process can invoke one of the provided adaptation methods with process templates as arguments. The composition agent then identifies the corresponding instance of that process and issues the adaptation method on it accordingly. The delegation process (see below) will use this option. As a second variant, an external process can invoke one of the provided adaptation methods with process instances as arguments. Here, a mapping is not necessary. This option is used by process $Userprocess_{\text{Consumer}}$ (section 4.1.4).

These two variants result in two different sets of adaptation methods. For the first variant, the process *CompositionAgent* provides the *same* adaptation methods as provided by the *Adaptationprocess*. To allow the direct adaptation of instances, a copy of the first set but with different method names and different parameters is provided. The process of a *CompositionAgent* can be formalized as follows (excerpt):

$$\begin{aligned}
CompositionAgent_{sid} &= TemplateMethods \mid InstanceMethods \mid HelpMethods \\
TemplateMethods &= !addBinding(aCompos_{sid}, A_{sid}.port_x, B_{sid}.port_y).B_{addBinding} \\
&\quad \mid !deleteBinding(aCompos_{sid}, A_{sid}.port_x, B_{sid}.port_y).B_{deleteBinding} \\
&\quad ... \\
InstanceMethods &= !addBindingInst(aCompos_{sid}, A_{sid}.port_x, B_{sid}.port_y).B_{addBindingInst} \\
&\quad \mid !deleteBindingInst(aCompos_{sid}, A_{sid}.port_x, B_{sid}.port_y).B_{deleteBindingInst} \\
&\quad ... \\
HelpMethods &= getInstance(Process, return).B_{getInstance}
\end{aligned}$$

Within the blocks B_{id} of the adaptation methods of process *TemplateMethods* it is assumed that the agent is able to identify the corresponding instances by mapping the passed component or composition specification to an instance. This mapping opera-

tion is indicated by method *getInstance* of process *HelpMethods*. This method takes a process template as an argument. Message *return* is a link that is used to send back the result (instance of that template) to the invoking process.

The Delegationprocess for delegating adaptation steps to instances

The *Delegationprocess* is responsible for delegating adaptation steps to the concrete running instances. This process is splitted into three separate processes, $Delegationprocess_L$, $Delegationprocess_R$, and $Delegationprocess_{Comp}$. The first is needed for delegating adaptation steps to all instances of a local composition of a peer service. The second one is used to forward adaptation steps to all instances of an interface composition and to delegate adaptation steps for changing the API of a peer service. The third process is used to delegate adaptation steps on a service composition, e.g. to add or delete a service or to alter the bindings between API ports. All three processes interact with the corresponding *CompositionAgent* processes. For the delegation of adaptation methods to all agents that are responsible for local composition (*CoL)* instances, the following process $Delegationsprocess_L$ is assumed:

$$Delegationprocess_L = start(currData)\left(\prod_{i=1}^{n} adaptmethod(info).CompositionAgent_{sid}^{i}\right)$$

Within this process, all *n* locally deployed *CompositionAgent* processes with respect to the peer service *sid* are placed in a sequence to pursue the adaptation method *adaptmethod* consecutively. The message *currData* has two submessages, *info* and *adaptmethod*, which are the necessary information used for the adaptation step specifying the name of an adaptation method, and more details about the affected components, compositions or ports. For the allocation of all remotely deployed *CompositionAgent* processes running on the behalf of the respective interface composition part *CoI*, the process $Delegationprocess_R$ must be modified slightly:

$$Delegationprocess_R = start(currData)\left(\prod_{i=1}^{n} \{adaptmethod(info).CompositionAgent\}_{Sid}^{Pid(i)}\right)$$

Here, the proxies of all *n CompositionAgent* processes with respect to the peer service *sid* are addressed that reside at remote peers under the address *Pid* (*Pid(i)* denotes the peer address of the *i*-th agent). The following process addresses the composition agent of a service composition:

$$Delegationprocess_{Compos} = start(currData)\left(\prod_{i=1}^{n} adaptmethod(info).CompositionAgent_{Compos-ID}^{i)}\right)$$

The operational semantic for invoking the process for delegating adaptation steps to a local or an interface composition is specified in this manner:

$$Userprocess_{Provider} \mid \underbrace{(start(currData).(adaptmethod_1.\cdots.adaptmethod_n))}_{DelegationsprocessL} \langle \phi$$

$$Userprocess_{Provider} \mid \underbrace{(start(currData).(\{adaptmethod_1\}.\cdots.\{adaptmethod_n\}))}_{DelegationsprocessR} \langle \phi$$

After the adaptation step has been carried out on a process template (section 4.1.1), the $Userprocess_{\text{Provider}}$ can react with either $Delegationprocess_L$ or $Delegationprocess_R$ along the *start* port of it. This depends on which composition part has been adapted by the user. During the reaction, the *currData* message is passed representing the adaptation method and all necessary data. This rule can only be chosen, if the adaptation methods are pursued on the level of a component composition. This restriction is ex-

pressed by the following side condition (let *ADAPTMETHODS* be the set of available adaptation methods; all arguments of function T_{process} are elements in *currData*):

$$\phi \equiv \begin{cases} adaptMethod \in ADAPTMETHODS \,/\{addPeerService, deletePeerService, \\ \qquad\qquad addBindingAPI, all\ auxiliary\ methods\}, \\ T_{process}(Composition_{sid}) = COMPOS\_SPEC\ (local\ or\ interface\ composition) \\ T_{process}(Component_{sid}) = COMP\_SPEC, Component\ is\ part\ of\ a\ composition \end{cases}$$

In general, all adaptation methods can be applied to a component composition except of methods for adapting a service composition and the auxiliary methods of Table 4-2.

The delegation of adaptation methods to a service composition is analogous to the delegation to a component-based composition but with a different side condition:

$$Userprocess_{Provider} \,|\, \underbrace{(start(currData).(adaptmethod_1.\cdots.adaptmethod_n))}_{DelegationsprocessCompos} \langle\, \phi$$

$$\phi \equiv \begin{cases} adaptMethod \in \{addPeerService, deletePeerService, addBindingAPI\} \\ T_{process}(ServiceCompos) = SERVICECOMP\_SPEC \vee \\ \qquad T_{process}(ServiceCompos) = COMPSERVICE\_SPEC \\ T_{process}(Service_{sid}) = SERVICE\_SPEC \end{cases}$$

Apparently, a service composition can only be adapted by means of methods on the level of a service composition. If a service is added to a composition, the *CompositionAgent* creates a new instance of it (by using system port **new**) and adds that new instance to the given instance of the service composition.

Auxiliary adaptation methods are not delegated to composition agents as they only apply within a local peer environment.

4.1.4 Tailoring an Interface of a Consumed Service

In order to tailor the interface composition of a peer service, a user interface is assumed allowing an actor to define and to pursue the necessary adaptation steps. These steps are *not forwarded* to the original provider peer. The primary intention is that these steps apply to the running instances. However, one can also think of having a local composition specification available so that one could also save the affected adaptations persistently. For covering this option, the process mainly responsible for tailoring a service interface ($Userprocess_{\text{Consumer}}$) has to react with the *CompositionAgent* to alter the composition or component instance directly:

$$Userprocess_{Consumer} \,|\, \underbrace{(!adaptationMethod_1.B_1 \,|\, ... \,|\, !adaptationMethod_n.B_n)}_{CompositionAgent} \langle\, \phi$$

If both processes are willing to react with each other, they have to satisfy the following side condition:

$$\phi \equiv \begin{cases} adaptMethod \in ADAPTMETHODS \,/\{addPortToFacade, deletePortFromFacade, \\ \qquad\qquad deleteBindingFacade, addBindingFacade, \\ \qquad\qquad addPeerService, deletePeerService, \\ \qquad\qquad addBindingAPI, auxiliary\ methods\} \\ T_{process}(Composition_{sid}) = COMPOSITION, Composition\ is\ a\ local\ composition \\ T_{process}(Component_{sid}) = COMPONENT, Component\ is\ part\ of\ a\ local\ composition \end{cases}$$

The side condition ensures that only instances of components and ports of a local composition can be altered. The modification of a façade is not allowed, because it would also affect the local composition. This could potentially lead to inconsistencies during the remote interaction between local and interface composition. Moreover, inconsistencies due to the erroneous adaptations of elements within an interface composition can also occur. The style presents no rules when adaptations are consistent so that they do not lead to abnormal behavior. To tackle this problem, for instance, integrity constraints could be entailed with respect to a composition. These conditions specify which adaptations steps are allowed to yield a consistent composition. Such constraints are described in the dissertation of Won [Won, 2004].

A user is also able to tailor a composite peer service. During runtime, he is able to dynamically integrate a new peer service or to define new bindings.

$$Userprocess_{Consumer} \mid \underbrace{(!adaptationMethod_1.B_1 \mid ... \mid !adaptationMethod_n.B_n)}_{CompositionAgent-Compos} \langle \phi$$

$$\phi \equiv \begin{cases} adaptMethod \in \{addPeerService, deletePeerService, \\ \qquad\qquad addBindingAPI, deleteBindingAPI\} \\ T_{process}(ServiceCompos) = COMPSERVICE \\ T_{process}(Service_{sid}) = PEERSERVICE \end{cases}$$

The degree of freedom to tailor a service composition depends on the execution model of the service composition. For the basic execution model, in fact no restriction needs to be made. For the distributed execution model, however, the candidate peer service to be tailored (i.e. the local part of it) must not be shared with other consumers ("one service per customer"). This ensures that no dependencies are violated to other consumer peers relying on the same peer service. Such rules should be declared in addition to the above side condition if necessary. The architectural style does not explicitly impose such constraints.

4.2 Integrity Constraints

The service composition model yet describes a structural template entailing which services need to be deployed and, thus, available for a composed application. Such a template can be seen as a *constraint* that defines a *state* for a given composition. The constraint is met, if all services are deployed and is not meet (or *violated*), if any of the defined services cannot be deployed. Apparently, if the constraint is violated, an exception can be presumed and a dedicated handler must be selected and executed. For each composition, this constraint is implicitly given. A component assembler is also able to explicitly add more constraints to a dedicated composition in order to describe valid states a composition has to fulfill during its runtime. A constraint that describes a valid state of a service composition is hereafter denoted as *integrity constraint*. An integrity constraint could, for instance, make assumptions about the importance of services in a composition. It could also configure, which of the bound services must be coercively available and which of these are less mandatory. Alternatively, a constraint could dictate that a composite service always has to be connected to a dedicated third-party peer service or at least to a given number of consumers.

Integrity constraints are interpreted as *contracts* that can be entailed between a consumer and a provider of a peer service. Contracts aim at improving the reliability of service compositions or composite peer services. These contracts are in particular im-

portant for scenarios, in which the peers form a collaboration in order to reach a common working goal (see motivation in section 2.4.1). The violation of a contract can occur due to the unavailability of a peer or peer service that is part of a contract. In this exceptional case, exception handlers can be defined that describe procedures how to handle the violation of that contract. A typical procedure thereby could be to locate, bind, and to deploy an alternative peer service adopting the role of the lost service.

Another assumption is that a contract (i.e. an integrity constraint) is associated to a dedicated *context*. A context could, for instance, be a working activity within a collaboration. Thus, many different integrity constraints can be defined on a service composition. However, only those integrity constraints need to be evaluated that are valid according to the current context. The context can be determined by a user or by the system. Both parties should be able to dynamically switch the context according to the goals, activities etc. of the given collaboration.

$SO_{P2P}A$ formalizes the notion of an integrity constraint only in a short way. The precise implementation of integrity constraints, context information, and exception handlers merely depend on the circumstances of the application domain in which a service-oriented architecture is deployed. DEEVOLVE, the reference implementation of $SO_{P2P}A$, describes the notion of integrity constraints in much more depth with respect to the application domain of networked collaborations in construction (see section 8.2 for the formulation of constraints and section 9.5 for an application scenario in which they are applied). During the next sections, only the base formalisms are presented.

4.2.1 Definition of Integrity Constraints

Informally, an integrity constraint consists of a *condition*, a list of additional *parameters* (specifying the target values for a condition), a *context* denoting when a constraint should be valid, and a *handler* realizing procedures to be executed if a condition is not meet). A condition represents a function that characterizes the integrity constraint. It can be assigned to a *condition level*. A condition level makes assumptions concerning the scope in which the condition can be applied. For instance, at component composition level, conditions can be formulated in terms of local components, ports and bindings between component ports (e.g. that a binding must be available). At a service composition level, conditions for integrated remote peer service can be formulated (e.g. that a certain peer service must be available, which could be entailed by a condition “mustBeAvailable”). At an architectural level, conditions could also comprise the availability of further peers or services (e.g. that consumers must be available for distinct peer service). A handler can be defined by the component assembler himself (see section 4.3). Alternatively, the default handler of a service provider can be taken if appropriate (section 3.3.10.5). As shown in section 4.3.2, a handler refers to a process consisting of many *options* that can be selected and executed upon detection of an exception. Based on the pi-calculus, the following definition of an integrity constraint can be formulated as follows:

Definition 4-1 (Integrity Constraint). Let $SC \in PossSC$ be a service composition (*PossSC* is the set of all available service compositions), $P = (param_1,\ldots param_n)$ a parameter list representing target values, $condition \in CONDITIONS$ a condition (set *CONDITIONS* represents all available conditions), $context \in CONTEXTS$ a context (set *CONTEXTS* represents all available contexts) and *Handler* an exception handler

($T_{process}$(*Handler*) = HANDLER_PROC) with internal *options*). Then the tuple *INT* = (*SC*, *P*, *condition*, *context*, *Handler*) is an integrity constraint for a composition *SC*.

A service composition *SC* can have many integrity constraints. All these are stored in process $IntegrityConstProcess_{SC}$. The *i*-th constraint of *SC* is obtained as follows:

$$IntegrityConstProcess_{SC} = !\left(\overline{getConstraint_i}\left(condition_i, paramlist_i, context_i, Handler_i\right)\right)$$

□

Process *IntegrityConsProcess* is attached to the process template of a service composition (Definition 3-18) as another parallel process. After the instantiation of a service composition, that process is still available and can be evaluated by other processes.

4.2.2 Evaluation of Integrity Constraints

In order to evaluate whether or not an integrity constraint is fulfilled, a separate process *CheckConstraint* is available. This process can be invoked along a single port *check*. It takes the condition, the composition process (*SC*), the context as well as the *target* and *actual* parameters as input. It is attached to the *Supplyprocess* of a peer environment and is addressable along the supply port:

$$Supplyprocess = !\, port_{supply} \mid CheckConstraints$$

$$CheckConstraints = !check(SC, context, condition, param_{actual}, param_{target} reply).B_{check}$$

The *CheckConstraints* process checks the integrity of the given composition (passed as a process) within the process bock B_{check} and returns the result (fufilled or violated) along the reply port. Thereby, it takes into account of current context. If the current context does not match the passed context of the integrity constraint, the constraint is not valid anyway. Process *CheckConstraints* can be used by any system process that can profit from the returned result. A process might in turn call an exception handler if the result is false, that is, if the integrity has been violated. The actual parameters represent parameters that have been gathered, for instance, by the *Controllerprocess* (section 3.3.11). Internally, the actual parameters are compared against the target parameters. The rule how to compare these two parameter set is dictated by the condition.

In general, the integrity of a composition can be evaluated in arbitrary intervals. However, one can limit the number of checking routines to dedicated events such as an occurred exception, that is, if a dependent (provider or consumer) peer has become unavailable. The actual parameters would then deliver the identifier of the failed service. Section 4.3.1 elaborates the checking of constraints as a reaction on an exception in more detail. Here, the *Controllerprocess* takes over the role to first check the integrity and then to invoke the appropriate handler if it is violated.

A useful process for checking the *n* predefined integrity constraints for a given service composition *SC* in succession can be defined as follows:

$$IntegrityCheckProcess_{SC} =$$

$$start(param_{target})\left(\prod_{i=1}^{n} \overline{port_{supply}}\left(check, SC\ , context, condition_i^{SC}, param_{actual}^{i}, param_{target}, reply\right)\right)$$

The process is initiated by invoking port *start* with message $param_{target}$, which represents the actual parameters that has been captured by some process. A further process is defined that checks the integrity constraints of all *n* service compositions in which a particular peer service *PS* has become a dependent service:

$$IntegrityCheckProcess_{PS} = start(param_{target})\left(\prod_{i=1}^{n} \overline{start}.IntegrityCheckprocess^{i}_{SC}\right) \langle \phi$$

The side condition for this process states that peer service *PS* is part of each service composition SC_i:

$$\phi \equiv \begin{cases} \forall\ SC_i : PS\ is\ part\ of\ SC_i,\ 1 \le i \le n \\ T_{process}(SC_i) = SERVICECOMPOSITION \\ T_{process}(PS) = PEERSERVICE \end{cases}$$

4.3 Exception Handling

Exception handling is important for an architecture that features fluctuating nodes (peers) with unpredictable behaviour. With respect to $SO_{P2P}A$, processes can fail or can become unavailable, which hereafter indicates an *exception*. Exception handling denotes the process of handling an exception in case it has caused the violation of an integrity constraint. Even if no integrity constraint has been formulated, handling an exception is necessary in order to avoid misbehaviour in service compositions or applications that rely on a failed peer service.

The process of handling exceptions is divided into two parts, *exception detection* and *exception resolution*. Exception detection incorporates the phases of detecting an exception by *monitoring* dependent peers and of determining if an exception has caused the violation of an integrity constraint. Exception resolution realizes a phase for selecting and executing the appropriate handler. As motivated in section 2.5.3.3, a user should actively be involved during that phase in order to apply the most suitable handler. A handler embraces *options* that make use of the adaptation methods proposed in the section 4.1.1. Thus, the same methods used for tailoring a service or service composition are also applied for resolving an exception. This eases users to comprehend the actions that are used to handle an exception and facilitates them to define own handlers. Users who are familiar with tailoring component-based software should therefore have few problems to develop and to work with exception handlers.

This *basic* phase model must clearly be refined for a concrete architecture. In particular, it must be evaluated, which types of users will be involved in exception handling and to what degree. The degree may vary depending on the competencies and incentives of the available stakeholders of an architecture. It also depends on the complexity of context information that needs to be taken into consideration. Especially in collaboration scenarios as outlined in section 2.4.1, such context information can necessarily influence the selection of a handler. Often, this context cannot be specified in advance. Here, users must be actively involved to assess the current context and, based on their awareness, to select the appropriate handler. If simple context information is available, the system could also take over the part to select and to execute a handler. This leads to an *adaptive system.* All in all one can say that the correct “mixture” between user involvement and an adaptive system must be traded off for each realization of a service-oriented peer-to-peer architecture.

A phase model could also illustrate, which users should be responsible to define exception handlers and integrity constraints. These aspects are not covered in the formal architectural style. Like the presentation of an integrity constraint, the purpose of

the next sections is to provide the basic formalisms of exception handling, only. More details on exception handling in a concrete architecture can be found in chapter 8.

4.3.1 Exception Detection

The basic mechanism for detecting an exception is to monitor all dependent remote peers on which a local peer dependents. This mechanism has already been described in section 3.3.11. Each peer environment has a *Controllerprocess* that is responsible for sending regular update messages to all dependent consumer peers that have subscribed for a peer service. In turn, each consumer peer is capable of returning its own status to the provider. Each *Controllerprocess* also maintains an internal list of *status ports* entailing the status of all dependent peer services. The process thereby maps the status of a peer to the respective status of a peer service. The port name includes the name of the peer service as well as the status of it (e.g. $peer_{sid}active$).

Before the remote interaction between two port facades can be carried out, a *Broker* process is responsible to query the current status of the peer service. If the peer service is active, the interaction can be executed (see operational semantics in section 3.3.11). If a dependent peer service obtains the status *failed* or *unavailable*, the broker of a port binding is not able to react with a port $peer_{sid}active$, the interaction cannot be performed. In this case, however, the broker process is able to react with the status ports denoting a failure of a peer (e.g. $peer_{sid}failed$). Instead of continuing the remote interaction, the *Controllerprocess* is responsible to execute the process of checking the integrity of the currently affected composition instance. If one of the imposed integrity constraints are violated with respect to the currently active context, the *Controllerprocess* activates the *Handlerprocess* to resolve the violation and, thus, to handle the exception. Both steps are explained in the following.

Process of Checking the Integrity of a Service

The failure, unavailability, or inconsistency of a dependent peer service can potentially violate predefined integrity constraints of local service compositions. Thus, it is necessary to validate integrity constrains after an exception has been detected by the *Controllerprocess*. The checking of a service composition assumes that the affected peer service has previously been aggregated with other services as a service composition. The operational semantics for checking the integrity is as follows:

$$\frac{\overline{PS_{sid}^{pid}\,status}\langle message\rangle.Broker_j^{id} \longrightarrow Broker_j^{id} \quad PS_{sid}^{pid}\,status(message).Controllerprocess \longrightarrow Controllerprocess}{ControllerProcess \mid IntegrityCheckProcess}\langle\phi$$

$$\xrightarrow{\tau} CheckIntegrity\ \ (infoPS_{sid}^{pid}.ControllerProcess, start.IntegrityCheckProcess_{SCsid})$$

An important side condition of the reduction rule states that the remote peer service PS_{sid}^{pid} is part of the local service composition SC :

$$\phi = \begin{cases} Peer\ Service\ PS_{sid}^{pid}\ is\ part\ of\ SC \\ status \in \{failed, unavailable, inconsistent\} \end{cases}$$

Again, only the status of consumed (remote) peer services can trigger the validation of the integrity constraints. Operation *CheckIntegrity* is defined in the following manner:

$$checkIntegrity\ (infoPS_i^{pname}(info), start(cuData)) \stackrel{def}{=} \nu c$$
$$\{c / infoPS_i^{pname}\}ControllerProcess_y \mid \{c / start\}\{info / cuData\}IntegrityCheckProcess_{SCid})$$

Message *info*, which is conveyed from the *Controllerprocess* to the *IntegrityCheck-Process* along the newly established link port *c* contains data describing the lost peer service. This data conforms to the actual parameters as expected and needed by he *IntegrityCheckProces*s (section 4.2). The *Controllerprocess* receives the result of the integrity check along the *reply* port. If the integrity of the service composition has been violated, the *Controllerprocess* activates the appropriate handler process being associated to the violated integrity constraint (section 4.3.2). If the status of the peer service is "failed" and no integrity constraint is violated, a default handler for handling the failure of a peer can be invoked instead (not further formalized here).

The integrity validation as formalized until now is arranged on an instance level: if a particular instance of a service is about to invoke a port of its implementation part, then all integrity constraints for the pertaining service composition are validated. Such a lazy validation prevents the composition from being inconsistent. In some application scenarios however, it would be more practical to check the integrity for all defined service compositions right after the status of a dependent consumer or provider peer has been changed. That is, the integrity should be checked, if the *ControllerProcess* gets a status update from a peer (invocation of methods *updateProvStatus* or *updateConsStatus*). For instance, the inconsistency or failure of dedicated consumers could potentially violate an integrity constraint associated with a provided peer service when it implies the permanent connection to a dedicated consumer. The operational semantics for checking the integrity due to a status update from a provider peer is modeled as follows:

$$\frac{PS_{sid}^{pid}\ status(message).Controllerprocess \longrightarrow Controllerprocess}{ControllerProcess \mid IntegrityCheckProcess} \langle \phi$$
$$\xrightarrow{\tau} CheckAllIntegrity\ (infoPS_{sid}^{pid}.ControllerProcess, start.IntegrityCheckProcess_{PSsid})$$

The side condition for this rule says that the peer service *PS* can either be a provided peer service being part of a composition or a peer service that depends on a local peer service:

$$\phi = \begin{cases} Peer\ Service\ SC_{sid}^{pid}\ is\ a\ consumer\ or\ a\ provided\ service \\ status \in \{failed, unavailable, inconsistent\} \end{cases}$$

A status update of a provider peer implies the validation of all local service compositions that directly depend on peer services provided by the updating provider peer. Operation *checkAllIntegrity* is defined as such:

$$checkAllIntegrity\ (infoPS_i^{pname}(info), start(cuData)) \stackrel{def}{=} \nu c$$
$$\{c / infoPS_i^{pname}\}ControllerProcess_y \mid \{c / start\}\{info / cuData\}IntegrityCheckProcess_{PSsid})$$

Note the difference between this operation and the operation *checkIntegrity* defined previously. While *checkIntegrity* invokes port *start* of the $IntegrityCheckProcess_{SCcid}$ process that contains the integrity constraints of a single software composition with identifier *cid*, *checkAllIntegrity* invokes port *start* of the $IntegrityCheckProcess_{PSsid}$ process that contains all software compositions including their integrity constraints that hold a dependency on the peer service *PS*.

4.3.2 Exception Resolution

If the *Controllerprocess* has detected the violation of an integrity constraint, it reacts with the *Handlerprocess* in order to resolve the exception. The *Handlerprocess* runs in parallel to the *Controllerprocess*:

$$ControllerProcess \mid Handlerprocess$$

The *Handlerprocess* consists of all *ExceptionHandler* processes that are associated to integrity constraints being part of local service compositions:

$$Handlerprocess = ExceptionHandler_1 \mid ... \mid ExceptionHandler_n$$

$$ExceptionHandler_i = handler_{cond}^{compos}(input).(Option^{System}\left(Option_1^{User} + ... + Option_n^{User}\right))$$

$$Option = (adaptation\ methods)$$

If the integrity constraint with identifier *cond* is violated within the currently running service composition *compos*, the *Controllerprocess* has to react with port $handler_{cond}^{compos}$ of the respective *ExceptionHandler* process. Message *input* delivers all necessary information about the occurred exception (e.g. the identifier of the peer service, semantic properties of it). Based on this input message, the process of handling the exception (i.e. the violation of the integrity constraint) can be executed.

An *ExceptionHandler* process has either been migrated by the provider peer (representing a default handler, see section 3.3.10.5) or has been defined by the consumer peer itself. Each handler potentially contains many *options*. Two types of options are available that can be executed, namely system or user options. *System options* are executed directly after invoking the respective *ExceptionHandler* process. Here, the user has no direct influence on the execution of that option. In addition, the user is capable of selecting additional options out of a set of *user options*. User options are composed with the *sum* operator ('+'). The application of the sum operator denotes a user decision point: at this point, it facilitates a user to select the most suitable option for a given application context. This selection process depends on the user's perspective and assessment of the current application context (see section 2.4.2.2 for examples of possible context data that could be taken into consideration). An exception handler can rely on both types of options or can only embrace a single type. Hence, an exception handler only possessing system options realizes a purely adaptive system. Again, the definition of an exception handler with system and user options must be traded off with respect to the conditions of the application domain, in which a concrete architecture will be deployed.

Options can use all adaptation methods that are defined in section 4.1.1. In the course of an option being part of an exception handler, these adaptation methods represent *actions* that can be executed. In order to handle, for instance, the failure of a peer service, options may refer to the methods "discoverPeerService" or "addPeerService" for integrating a new service. Section 8.4.2 outlines a concrete set of adaptation methods that realize actions for handling exceptions in the DEEVOLVE environment.

4.3.3 Exception Cascading

The *Controllerprocess* is also capable of delegating an exceptional case (i.e. the failure of a dependent peer) as well as the event of an integrity violation to further peer

processes. The process of *exception cascading* allows other peers to issue their own exception handling processes as well. For delegating exceptions, the *Controllerprocess* needs to be extended as follows:

$$Supplyprocess = Controllerprocess \mid Cascadeprocess$$

$$Cascadeprocess = delegateException(porty_{supply}, info, type).B_{cascade}$$

$$Controllerprocess = ... \mid receiveException(info, type).B_{receiveExcep}$$

If the local *Controllerprocess* detects an exception or an integrity violation, it is able to react with the *Cascadeprocess* in order to delegate these occurrences to a peer. A peer is addressed along its supply port, which has been stored internally after the subscription process (section 3.3.10.6). The sub process B_{cascade} performs the actual delegation by addressing the port *receiveException* that is part of the *Controllerprocess* of the remote target peer. Message *info* thereby holds information on an exception (e.g. the name of a failed peer), whereas message *type* denotes the type of exception (e.g. failure, or integrity violation). By receiving a message along this port, the remote *Controllerprocess* is able to perform a local integrity check as explained in section 4.3.1. If the remote exception also violates a local integrity constraint, the exception resolution process is activated (section 4.3.2).

By default, the *Controllerprocess* could forward the exception to *all* subscribed peers. A refinement would be to address only those peers that have evinced interest in the cascading of an exception. In this case, the *Controllerprocess* has to implement a dispatcher process in order to select the relevant peers out of the list of all subscrived peers. The information, if a peer is interested in receiving messages on occurred exceptions, can be signalized as a further attribute during the subscription process (see section 3.3.10.6 for information).

The process of exception cascading is useful for integrity constraints that incorporate several peers that, for instance, collaborate in a sequence according to common process or workflow model. Here, the failure of a transitively connected peer can necessarily lead to an integrity violation, if the local peer expects the availability of all peers within a given process chain. A concrete example for such an integrity constraint is given in section 8.2.4 (Information Flow Integrity).

4.4 Related Work and Scope

This section summarizes related work that compares the formalization approach of a service-oriented peer-to-peer architecture with other correlated approaches. The following subsections list external works according to several topics. They compare the results of chapter 3 and 4, respectively.

Type system and general aspects of pi-calculus

As already stated in section 3.1, the $SO_{P2P}A$ architectural style is merely based on the pi-calculus, a process calculus for formalizing distributed software systems. The original pi-calculus of Milner ([Milner, 1991]) assumes that processes can transport arbitrary messages (with respect to Milner's work so-called *names*) through channels. Names can represent either data or links (i.e. references) to other processes. Despite of this obvious division, there is no dedicated type system that gives means to both channels and names. The only implicit typing rule states that only ports of different orientation (in- vs. out-channels) can react with each other.

This work proposed a type system in order to distinguish between simple data values, links to processes, and processes that could potentially be conveyed through ports. Moreover, processes themselves can be associated with types and predicates for precisely specifying their responsibility (e.g. components, peer services, user interface processes). A plethora of further type systems have been proposed for the pi-calculus. Many type systems aim at describing the capabilities of channels. Pierce and Sanigiorgi propose a type system in order to obtain read-only, write-only, and read-write channels [Pierce and Sanigiorgi, 1993]. Kobayashi et al. introduce a type system which can be applied for restricting the number a channel can be used [Kobayashi *et al.*, 1999]. Data types for names can be found in the work of Pahl [Pahl, 2001], who also introduces the notion of channel types to describe what kind of data can be transported through a channel. Type systems for describing the nature of processes could not be found.

Formalizing Component-based Architectures with the pi-calculus

A number of different approaches can be found for formalizing component-based architectures based on the pi-calculus. Nierstrasz and Achermann [Nierstrasz and Achermann, 2003] propose the PICCOLA calculus, a high-level calculus for formalizing software components based on the asynchronous variant of the pi-calculus [Boudol, 1992]. The asynchronous pi-calculus allows for the asynchronous, that is, non-blocking communication between processes. Asynchronous communication has certainly a closer and more intuitive association to distributed computing in general. It would clearly benefit the $SO_{P2P}A$ architectural style as well. It is, however, not the idea of $SO_{P2P}A$ to propose the most effective communication patterns for service-oriented peer-to-peer architectures, but to propose a flexible architecture allowing for adapting component-based services and service compositions in case of exceptions. PICCOLA also involves the idea of having generic adaptors for composing incompatible components. Adopters are not provided by $SO_{P2P}A$ but can be simulated by local components for mediating between incompatible service interfaces. Finally, the PICCOLA calculus serves as the semantic foundation for the PICCOLA language (dissertation by [Lumpe, 1999], a concrete language for composing software compositions. An implementation of this language is available in Java (JPICCOLA) allowing for composing Java components. In this work, the $SO_{P2P}A$ style serves as the foundation for the PeerCAT composition language (from section 6.4.3). In a nutshell, the formal component model of Nierstrasz et al. is a generic model suitable for many composition styles. Especially for distributed composition styles (client-server, peer-to-peer, etc.) adequate formalisms for detecting and handling failures are missing. Those formalisms – in particular presented throughout this chapter – constitute the strengths of this work compared to other works.

The work of Pahl [Pahl, 2001] introduces a variant of the pi-calculus for modeling the composition of components and the runtime replacement of components from an existing composition. Analogously to the $SO_{P2P}A$ style, Pahl's work assumes a component port model, where each port is made up of three different sub ports, namely contract, interaction, and (if necessary) reply port. He also presumes that two contract ports first have to react before any service can be invoked through an interaction port. The operational semantics are defined only for the interaction between various components in a (sound) composition. There is, however, no information concerning the actual structure of a component, only the interface is modeled in terms of a component life-cycle. No internal semantic is defined as opposed to $SO_{P2P}A$ (see section 3.3.5). Besides, Pahl introduces formalism for static (i.e. during design) and dynamic (i.e.

during runtime) replacement of components. He introduces a construct that allows for changing the type of a port during runtime as the only operation for adapting a process expression. In the $SO_{P2P}A$ style, more concise adaptation operations are introduced that are familiar to typical component-based construction methods (adding and deleting of behaviour, changing connections). What in general remains unclear in Pahl's approach is the question, *who* or *what* is the actor for carrying out the adaptation of a composition process. Also, there is no information who is responsible for triggering the initial composition of components. In order to clarify these questions, $SO_{P2P}A$ has established various process types that take over the role of a trigger to initiate adaptation and composition operations. This can either be a system process (process type SYSTEMPROCESS) for the initial composition or a user interface process (type USERPROCESS) for adapting (tailoring) a composition from a user perspective.

Formalizing Distributed Software Architectures with the pi-calculus

Amadio presents an extension to the pi-calculus for distributed computation [Amadio, 2000]. He enriches the calculus with features for the explicit distribution of processes to locations, routing of messages, mobility of processes, and the failure of locations and their detection. For the detection of failures, he introduces a special channel (**ping**) that allows for monitoring remote locations. Further special channels are formalized for process migration (**spawn**) and for stopping a remote process (**stop**). All three channels are part of a location process that is similar to the *Peercontroller* process within the peer process (section 4.3). Although the approach favours the handling of exceptions as a central contribution, there are no handler formalisms available for resolving an exception. Compared to $SO_{P2P}A$, the approach lacks on some more convertible and practical formalisms. $SO_{P2P}A$, for instance, introduces the *Userprocess* as an interface to (human) actor whenever decision-making is necessary. Such concept is missing in Amadio's work. The strength of his work lies in the formal presentation of tools for reasoning about the equivalence and bisimilarity of processes. Such tools are out of scope of this dissertation and have been omitted.

Chothia and Stark present an extension to the pi-calculus that allows for creating local areas in which processes can interact [Chothia and Stark, 2001a] in an insular way. Areas are arranged in a hierarchy of levels, distinguishing for example between a single application or a host. This approach could potentially be adopted for simulating the peer group approach by $SO_{P2P}A$. By means of a peer group, peers (= processes) are capable of interacting in a self-contained way as well. In contrast to $SO_{P2P}A$, the approach of Chothia and Stark defines no way how to create, maintain, and join an area.

In the dissertation project of Borgström [Borgström, 2003] [Borgström *et al.*, 2004], the author proposes an extension of the pi-calculus to formally model a distributed hash table (DHT) for peer-to-peer overlay network. DHTs are used for realizing the routing mechanisms within such networks. The proposed model is used for verifying the functionality of a DHT. This is done by proving the correctness of typical operations of a DHT such as the lookup operator (see [Borgström *et al.*, 2004]). In contrast to the $SO_{P2P}A$, it is not the intention of Borgström's work to specify services, service compositions, service adaptations, and the functionality of a peer environment. Effective and correct routing mechanisms are somewhat important also for service-oriented peer-to-peer architectures. In the present work, these mechanisms are only formalized in a rudimentary way (see section 3.3.8). Borgstrom's calculus could therefore enrich the $SO_{P2P}A$ style at this point.

Pi-calculus for Modeling Workflows

The pi-calculus has recently exerted influence on the first version of the Web Services Choreography Description Language (WS CDL) proposed by the W3C [W3C, 2004d]. WS CDL intends to serve as the default standard notation for modeling choreography workflow compositions based on Web Services standards (WSDL, SOAP). Partners within a composition then interact in a peer-to-peer style. Despite the expressivity of WS CDL, the language is far away from the maturity of the $SO_{P2P}A$ style. Above all, no assumptions are made concerning the implementation of single service. In addition, mechanisms for exception handling have not been formalized accurately (only in a brief textual way, see section 2.4.8). The influence of WS CDL in industry can, to date, not exactly be estimated due to the early status of the proposal.

Adding Formal Semantics to the UML

Chapter 3 merely used visual diagram from the UML (v2.0) notation for elucidating the notions of components (by means of component diagrams) and component compositions (by means of composite structure diagrams). This approach has not only improved the comprehension of the complex formalisms, say, for unskilled readers. It has also provided exact formal semantics to both diagram types. This is clearly an improvement compared to the conventional UML semantics, which are expressed in terms of a semi-formal meta-model. This meta-model is based on natural language descriptions. A couple of approaches for formalizing UML diagrams can be found (e.g. by means of Z, temporal logic). Mwaluseke provides a good survey on existing approaches [Mwaluseke and Bowen, 2001]. From the range of structural diagrams in UML, merely class diagrams have been enriched by formal semantics. At the time of preparing this dissertation, no approaches for formally specifying the semantics of component diagrams and composite structure diagrams could be found. The dissertation therefore claims the formal description of semantics for both component and composite structure diagrams as an other important contribution to the state of the art.

Other Algebras for Formalizing Component-based Software Architectures

Various other algebras have been used for formalizing component-based software architectures. Barbosa uses the *coalgebra theory* as the solid foundation for formalizing software components that provide some interface and maintain some internal state [Barbosa, 2000]. He also provides formalisms for composing components to compositions. The coalgebra theory is a somewhat more complex algebra than process algebras in general and, therefore, less suited for a later implementation towards a concrete software architecture. Barbosa's approach therefore remains rather theoretic. No formalisms are provided for adapting components. Malcom offers a similar approach for modeling components by means of *hidden algebra* [Malcom, 2005]. A (quick) review of this paper revealed a huge formal complexity of the approach. Aspects covered in this dissertation have not been considered.

In the dissertation of Oliver Stiemerling [Stiemerling, 2000], the author has formalized a component-based client-server architecture by means of the data spaces theory [Cremers and Hibbard, 1978] [Cremers and Hibbard, 1986]. Besides the formal structure of a distributed application, he also formalized adaptation operations for tailoring such distributed application. He could prove that an adaptation always yields a correct composition with respect to the data space theory.

Apparently, all mentioned algebras provide solid mathematical foundation for rigorously formalizing a distributed software architecture style such as $SO_{P2P}A$. The justi-

fication for the adoption of the pi-calculus can be attributed by the recent popularity of this calculus in the area of workflow modeling. Moreover, the calculus seems to be sufficiently intuitive to comprehend and, more importantly, to be applicable and convertible into a concrete software architecture.

Runtime Analysis of Dependencies

The approach for analyzing consumer dependencies takes into account only static information in terms of user data that has been received by consumers during subscription. It does, however, not consider dynamic runtime dependencies between instances of peer services. Such dependencies potentially might exist when peer services are used by consumers. The checking of dynamic runtime dependencies is actually out of scope for this work. For completeness, an adequate condition to ensure the non-availability of dynamic dependencies had to be inserted in the side conditions when invoking the adaptation methods of process *Adaptationprocess* (from section *4.1.2 and 4.1.3*). For instance, for the side condition of section *4.1.3*:

$$\phi \equiv \begin{cases} adaptMethod \in ADAPTMETHODS \,/\{discoverPeerService, addPortToFacade...\} \\ T_{process}(Composition_{sid}) = COMPOS\_SPEC, Composition\ is\ a\ local\ composition \\ T_{process}(Component_{sid}) = COMP\_SPEC, Component\ is\ part\ of\ a\ local\ composition \\ No\ runtime\ dependencies\ available \end{cases}$$

This additional condition is certainly somewhat fuzzy and does not entail which method or model is actually beyond this condition. Once again, this work has not focused on the aspect of dynamic adaptation of components. The interested reader should refer to the very good dissertation of Pascal Costanza for obtaining a sound overview on the state of the art on this research field [Costanza, 2004].

Transactional Behavior

Research from the fields of component-based adaptation and component-based runtime evolution often arises the question concerning *transactional* behavior. Transactional behavior has become a popular way for ensuring data integrity in databases. From that area, transaction properties such as the ACID property have been proposed to guarantee this requirement. In the context of this work, a transaction could ensure that *all* instances of a peer service are adapted consistently. If the adaptation of at least one instance failed, the complete adaptation process would be rejected (rollback), that is, *no* instance would be adapted. This would then lead *again* to a consistent state. In a peer-to-peer architecture, this – at first glance straightforward – model becomes more complex. The adaptation of an instance could also stimulate affected consumers to adapt their dependent provided services as well. In the case of a rollback as mentioned above, there is a transitive chain across several peers that needs to be roll-backed.

In point of fact, this work does not fully address this problem of transactional behavior. In the context of service-oriented architectures, this problem has recently been recognized (see [Farnoudi, 2006] for a good overview). Farnoudi also discusses possible ways for handling transactions in service-oriented peer-to-peer architectures. However, no groundbreaking approaches are known at the time of writing this work.

Chapter 5
Assessment of FREEVOLVE's Concepts

The purpose of this chapter is to assess state-of-the-art concepts of the FREEVOLVE client-server architecture [Stiemerling, 2000] [Won, 2004] for the *adoption* towards a service-oriented peer-to-peer architecture. Taking the concepts of this platform into consideration makes sense since adaptation strategies have been examined with respect to the special requirements of a distributed client-server architectural style. Like in the $SO_{P2P}A$ style, component-based adaptation (*tailoring*) strategies have been proposed serving as an efficient fundament to build tailoring environments dedicated for end-users. This chapter briefly summarizes the concepts of FREEVOLVE. It then presents an in-depth discussion, to what extent the component model, the structural model of a distributed application, and the adaptation strategies for a client-server architecture can be adopted to meet the requirements of the $SO_{P2P}A$ architectural style.

5.1 Core concepts of FREEVOLVE

FREEVOLVE is a runtime environment for the deployment of *component-based client-server applications*. The runtime environment is actually spread over a dedicated FREEVOLVE server and (potentially several) FREEVOLVE clients. All applications that can be deployed and executed within this distributed runtime environment are made of compositions of single components. Initially, both client-sided and server-sided components reside on the FREEVOLVE server. A single client can request a list of available applications, hosted on a single server. Having chosen a distinct application, all constituting components for the client part of that application are migrated, instantiated, and eventually executed in the client's runtime environment. A client must have been registered in the user management of a FREEVOLVE server, before it can access the hosted applications of that server. The operator of an FREEVOLVE server is capable of assigning roles to an enrolled client. A role can confine the number of applications a client is able to access. At any time, a client is only able to address exactly one server. That is, a switch to another server cannot be arranged during runtime.

5.1.1 The FLEXIBEAN Component Model

FREEVOLVE incorporates the FLEXIBEAN component model for specifying both *structure* and possible *interaction primitives* for components. This model is an extension to the conventional JAVABEANS model developed by Sun [Sun, 2000]. Most notably, it accomplishes the remote interaction between client and server components through the

explicit integration of the Java Remote Method Invocation (Java RMI) technology, which is part of the standard edition of Sun's Java platform (J2SE). All other important concepts of the FLEXIBEANS model are summarized in the following:

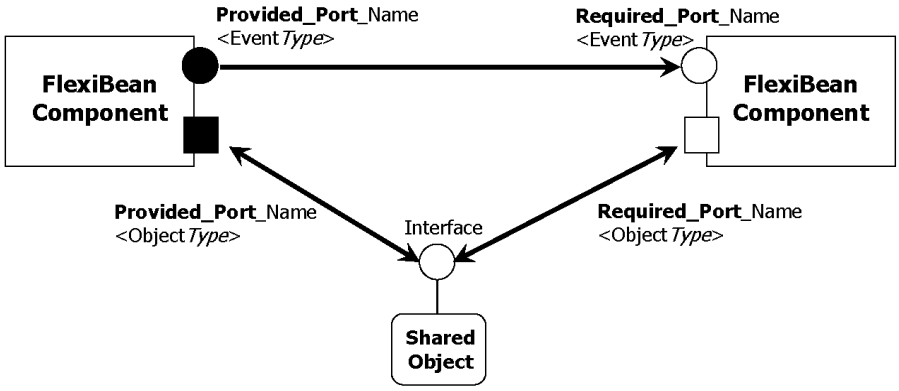


Figure 5-1: The FLEXIBEAN component model (notation based on Stiemerling's dissertation)

- In analogy to the JAVABEANS model, FLEXIBEANS provides a uni-directional and synchronous *event notification* primitive. Event notification follows the observer pattern: an event source produces an event, which is passed to one or many event sinks. Due to the uni-directional character of event notification, an event sink does not send back a confirmation event to the source (see [Buschmann, 1996] for details concerning this pattern).
- Unlike the JAVABEANS model, the FLEXIBEANS introduces a *shared object* as another interaction primitive. A shared object is an object that is shared between two components, whereas both components have common access on it. In contrast to the uni-directional event flow realized by the event notification primitive, a shared object can simulate a bi-directional data flow between components.
- The interface of a FLEXIBEANS component is represented by *typed* and *named ports* (cf. Figure 5-1). Ports serve as the connection point of a component. A port can act either as a *provided* or as a *required* port. For event notification, an event source is represented by a provided port, while an event sink is indicated as a required port. For a shared object, a provided port initially provides the shared object, which is passed to the required port belonging to the corresponding component. Connecting two ports assumes type-equality, that is, both ports must provide or require the same event object or shared object, respectively. In addition to types, ports can hold names in order to distinguish between type-equal ports.

The composition of FLEXIBEANS components towards concrete (client-server) applications can be formulated in a *declarative* way by the CAT (*Component Abstract Template*) architecture description language. The task of CAT is to identify all components appendant to a composition and to define bindings among the pertaining ports of these components (*horizontal binding*). CAT features a *hierarchical composition* of components: so-called *instance components* (correspond to single class-files) can be further composed to *abstract components*. This way, a composition of components yields, again, to a single component. Although an abstract component can itself feature ports, it does not commensurate to a class-file. Instead, abstract ports are bound to concrete ports belonging to instance components (*vertical binding*). In each case, the clients as well as the server part of an application are implemented as abstract components. The resulting whole client-server application is indicated as a *system component*.

Figure 5-2 visualizes the structure of the FREEVOLVE runtime environment. The client and server compositions (abstract components) are described in the ClientCAT and ServerCAT files, respectively. Another CAT file (DCAT) is used to specify the system component by declaring the remote interactions between the abstract component of the client and the abstract component of the server. If the user of a FREEVOLVE client has requested an application to be executed in his environment, not only the components, but also an object-oriented presentation of ClientCAT structures (*proxies,* see section 5.1.3) are migrated to the client. Both the instantiation and the binding of all components are carried out by FREEVOLVE with respect to the ClientCAT definitions.

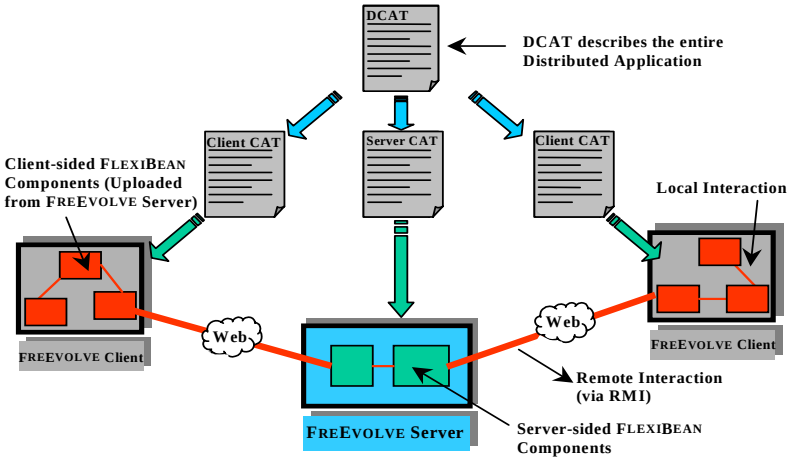


Figure 5-2: Structure of the FREEVOLVE runtime environment

5.1.2 Tailoring Components

The CAT language as presented in the previous section allows for the declarative composition of components. During the process of deployment, the compositional description is interpreted and the appropriate composition routines are executed. Basic composition routines are, in particular, the *binding of ports* and the *setting of predefined values* for attributes[18]. The actual goal of FREEVOLVE is not only to provide a runtime environment for the deployment and execution of component-based applications but also to provide new concepts for the flexible adaptation of deployed compositions during runtime. Particularly less skilled end-users should be enabled to adapt compositions according to their personal requirements and needs. As mentioned in the state of the art section (2.1.2.2), end-user adaptation is also referred as *tailorability*.

According to the notion of *component-based tailorability,* the same routines as obvious for the composition of components (binding components, adding or deleting components, setting attributes) are adopted for tailoring assembled components. A perspicuous distinction between tailoring and composing components can certainly not be established, as each notion points out a way of programming applications in a *declarative* manner. In the context of FREEVOLVE, a discrimination between these two notions is made with respect to the point in time, when a composition is tailored or composed, respectively. While the composition of components is usually carried out

[18] In FREEVOLVE, attributes represent information concerning the placement of visible components within a container. A FLEXIBEANS component cannot define nor modify these attributes directly, as they are only part of the compositional description of a composition.

during design phase of an application, the tailoring of components denotes the subsequent modification of a composed and already deployed application during runtime.

The FREEVOLVE runtime environment provides an Application Programmable Interface (TAILORING API) encapsulating necessary routines for tailoring components. Tailoring routines comprise the creation and deletion of bindings between ports of components, the addition and deletion of components, the addition and deletion of single ports as well as the modification of values of attributes. Various tools such as the TAILORCLIENT [Krüger, 2002] or the 3D-CLIENT [Hallenberger, 2000] have been developed implementing the TAILORING API. All mentioned tools enable end-users to flexibly tailor both client and sever components in a graphical manner.

The competence of an end-user for tailoring components dependents on his assigned role within a given FREEVOLVE environment. Apparently, not all end-users should be allowed, for instance, to tailor server components, as even small modifications could lead to inconsistencies with running client components. On the other hand, authorized users are permitted to tailor client-sided components, where the effected tailoring routines do modify the components of *all clients* at runtime. Any modifications to client and server components are also stored persistently within the respective CAT files. The task of both effecting tailoring routines to components and storing modifications to disk is thereby fulfilled by *proxy objects*. The purpose of proxy objects will be elaborated in more detail in the forthcoming section.

5.1.3 Proxy Objects

In order to maintain a component-based composition during runtime, a parallel object structure is instantiated consisting of *component* and *proxy* objects (also denoted as '*P*' objects, components as '*C*' objects). '*C*' objects are generated from all available CLIENTCAT and SERVERCAT descriptions during startup of a FREEVOLVE server. A single '*C*' object represents a template of an abstract or an instance component with respect to the CAT description. Proxy objects are representatives of concrete instances of these components within the FREEVOLVE environment *at runtime* (Figure 5-3).

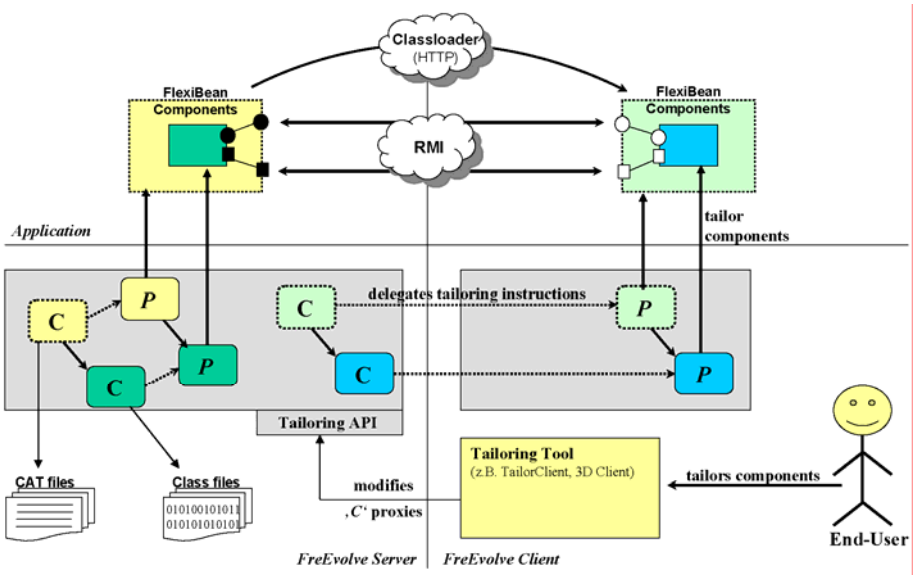


Figure 5-3: Proxy structure of FREEVOLVE is responsible for the actual management of client and server components

Since CAT supports hierarchical composition, a tree data structure can be derived out of these objects, consisting of inner nodes (indicating abstract components) and leaves (representing instance components).

If a user requests for the execution of an application, copies of these '*C*' objects (i.e. '*P*' proxies) are made, that, henceforth, take over the control of the related components. Proxies representing server components remain in the server's runtime environment, while proxies for client components are migrated to the client's environment. At start-up of an application, '*P*' proxies are responsible to instantiate all components in the respective (client or server) runtime environment and to establish all declared bindings among them. This way, a '*C*' object tree can be assigned to potential many '*P*' proxy trees. The TAILORING API of FREEVOLVE affords the modification of the '*C*' data structures. Arbitrary tailoring routines emitted by users are forwarded to all '*P*' structures, which eventually apply them to their controlled components. Tailoring '*C*' structures of client components brings forth that appropriate tailoring routines are invoked in all running client environments hosting client components. Besides broadcasting tailoring routines to '*P*' structures, '*C*' structures are also responsible to modify the pertaining CAT files, so that any tailoring activity is stored persistently to disk.

The *structure of a distributed application* in FREEVOLVE can be depicted by an UML class diagram [Stiemerling, 2000]. A structural diagram encompasses a user-oriented view on a distributed application, the underlying hierarchical organization of component and proxy structures, the remote binding, and the relation between component and proxy structures (see Figure 5-4).

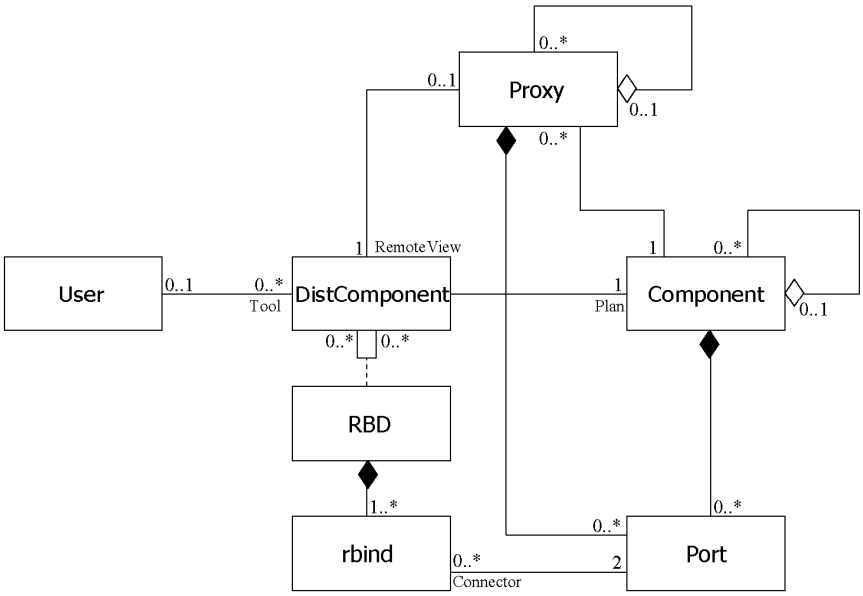


Figure 5-4: UML class diagram for the representation of a distributed system structure

According to Stiemerling's work, class "DistComponent" represents a coherent view on either a client or a server composition. Each composition is described by a component tree (plan). A concrete instance of composition is described by a proxy tree. If an object of class "DistComponent" refers to a client composition, then this object must be associated to a specific user object. In this case, the "DistComponent" points to the proxy structure that is located at the client's local FREEVOLVE runtime environment.

The remote binding between two distinct client and server compositions is modelled by the association class "RBD". This class is the composite part of a composition hi-

erarchy with a number of so-called "rbinds" classes (remote binds) as the part structure. Each rbind object defines a link between two ports pertaining to a component.

5.1.4 Prototypical Implementation and Applications

The prototypical implementation of the FREEVOLVE runtime environment as well as corresponding tailoring tools has been realized by a couple of master theses projects (in particular [Hinken, 1999], [Hallenberger, 2000], [Krüger, 2002]). The current implementation is entirely based on the Java 2 platform. Apart from the recent popularity of Java, the strong relationship to Java can be justified by the adoption of the Java-oriented FLEXIBEANS component model in FREEVOLVE. All other concepts outlined in the previous sections are language-independent, that is, the CAT composition language, the structural model of a distributed application and the component-based tailoring strategies. Given a component model incorporating alternative techniques for remote interaction (such as CORBA, SOAP, JXTA's pipe protocol) an implementation of FREEVOLVE in a different object-oriented language would be feasible as well.

5.2 Additional Concepts

In this section, additional concepts of FREEVOLVE are presented that are beyond the original concepts conceived by Stiemerling. The concepts of server sessions (5.2.1) and semantic integrity concepts (5.2.2) are essential for the forthcoming discussion of adopting FREEVOLVE towards a peer-to-peer runtime environment.

5.2.1 Server Sessions

The notion of *server sessions* [Alda *et al.*, 2002a], [Alda *et al.*, 2002b] constitutes the first step to enhance FREEVOLVE towards a peer-to-peer runtime environment. According to this notion, exact copies of the server components belonging to a distinct application can be migrated to arbitrary client environments. Server components are deployed and executed as (remote) server sessions in client runtime environments. Any client being enrolled in the original FREEVOLVE server is also capable of interacting with a server session. Note, that FREEVOLVE's original distinction between client and server is *almost* revealed: each client environment is not only accomplished to use, but also to *host* server components being accessible for other clients. In addition, client environments are enabled to directly interact with each other without the interplay of a server environment.

Each client is able to obtain a list of all running server sessions within a given FREEVOLVE environment. The pertaining user can then select an individual session among the retrieved sessions. Again, a role model can restrict the number of server sessions a particular user can obtain and, thus, the possible session he can interact with. For each migrated server session, the corresponding '*P*' proxies are also migrated to the hosting client. If a user tailors the '*C*' structure of a server application on the FREEVOLVE server, the related remote '*P*' structures are notified about the effected tailoring routines. The '*P*' proxies are then instructed to directly apply these tailoring routines to the components forming the server sessions.

With the concept of server session it is possible to divide the complete number of FREEVOLVE client environments into *subgroups*. Each subgroup is governed by a sin-

gle client hosting the server session. Only authorized clients are able to join a subgroup and to interact with other clients enrolled in the subgroup.

The benefit of FREEVOLVE supplemented by the server session concept is elaborated in conjunction with the support of decentral organizations in civil and building engineering projects in [Alda *et al.*, 2002a] and [Alda *et al.*, 2002b]. Furthermore, FREEVOLVE has been adopted as the fundamental architecture for a distributed chat system aimed to support collaborative learning processes in the context of Geoinformation systems [Bode *et al.*, 2004]. Here, server sessions are used to model chat sessions among a confined number of persons.

5.2.2 Semantic Integrity Conditions

Semantic integrity constraints for component-based compositions have been studied fundamentally in the dissertation of Markus Won [Won, 2004]. Integrity conditions are used to restrict the number of possible tailoring operations on a component-based composition. This approach especially assists unskilled or novice end-users during tailoring their local applications. The violation of a constraint occurs through the uncontrolled adaptation of a component artifact. Depending on the chosen integrity constraint, users obtain a feedback report or an alternative adaptation is proposed.

Integrity conditions denote a state of a composition. Each integrity constraint consists of a constraint, which is expressed as a function *r*. Each function takes a set of parameters *P*. The application of a function checks whether a certain state of a composition is fulfilled or not (*CAS* is the union of sets of all possible component systems):

$$r : CAS \times P \rightarrow \{true, false\}$$

The set of parameters *P* thereby points out the expected state. Won has formalized a set of different constraint functions *r*, whereas each function is associated to a unique constraint level. Constraint levels are components, parameters, and ports (see [Won, 2004], p. 89, table 9 for an overview).

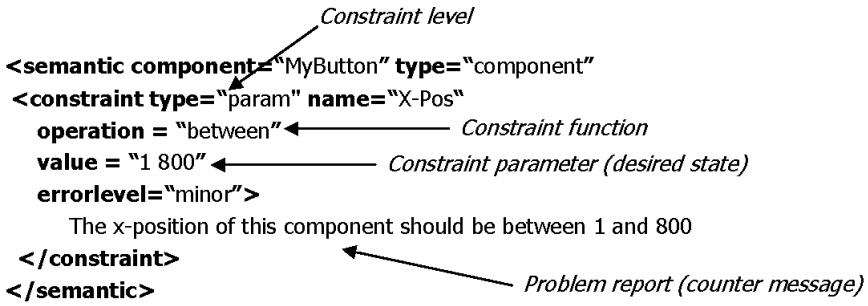


Figure 5-5: Example for a basic constraint expression in XSemL

In order to apply such constraints to a declarative CAT composition, the constraint language XSemL has been developed. This language allows for specifying integrity constraints based on the meta-language XML [W3C, 2004a]. The example in Figure 5-5 demonstrates the usage of XSemL to define a constraint on parameter level. This integrity constraint implies that the position of a component (parameter X-Pos) should be in the range of 1 and 800 (desired state of component). Each time, an end-user carries out an adaptation on this component, this constraint is checked. The constraint is violated if an end-user puts the component outside of the desired range (e.g. 1000). In this case, a problem report is displayed that contains a counter message.

Won also proposed a refined integrity type, the so-called *solution integrity*, in which more expressive routines (solutions) for handling exceptions can be produced. These routines correspond to the same operations as for tailoring a composition (see [Won, 2004], p. 92, table 10 for an overview). An action part for the basic constraint expression in Figure 5-5 can be formulated as follows (Figure 5-6):

```
<action command="setparam" type="component"
   <param name="name" value="X-Pos"/>
   <param name="value" value="400"/>
</action>
```

Figure 5-6: Action part to handle an exception (XSemL expression)

In case of a defective tailoring step where the expected state of the *MyButton* component is violated, the action *setparam* is invoked. This action sets the value of parameter *X-Pos* back to an initial and correct value.

Note that the declarative expressions of XSemL only specify the state of a component and pertaining actions for handling the violation of a state. However, they do not implement any kind of algorithm or process to check the correctness of an integrity constraint. This checking process is implemented by additional classes (Java code) that are implicitly augmented to an integrity constraint (see [Krüger, 2002] for more details on the implementation of the integrity concept).

5.3 Assessment of the presented concepts

The purpose of this section is to assess the outlined concepts of FREEVOLVE concerning a possible portability to a peer-to-peer runtime environment. It will be discussed, which of the presented concepts can be adopted and which of them have to be revised.

5.3.1 Component Model and Component Composition

The FLEXIBEAN component model describes components that feature typed and named ports as public access points. Ports of different components can be connected if they are type equal and hold a different polarity (required to provided ports). This model is in accordance to the port model of $SO_{P2P}A$. There is, however, a subtle difference between the polarity concepts between the FLEXIBEAN component model and the component model of $SO_{P2P}A$. In FLEXIBEAN, the provider of a port marks a component responsible for producing an event and for sending it to its connected (subscribed) required ports. It does so by invoking specific methods implemented by the component possessing the required ports (see [Stiemerling, 2000]). Following the $SO_{P2P}A$ style, a port provider represents a component offering an interface that can be used by required ports. The event-notification concept can though be simulated with the component model in this work, because a required port can hold references to many provided ports. With that, a required port can notify many (subscribed) provided ports at the same time following the underlying observer pattern of event-notification. The association of polarity information to ports then needs to be adjusted.

The shared object interaction primitive has not been conceptualized by $SO_{P2P}A$ directly. However, this primitive can actually be used to actualize the general port pattern of the proposed architectural style. The provider of shared object corresponds to

the component offering an interface that can be used by other required ports. The polarity concept could be used without modification.

Abstract components could serve as a way for realizing the facades of composition parts. Here, the FLEXIBEAN model is even more powerful as it allows for modelling components in a hierarchy with an arbitrary depth. Owing to the remote interaction capabilities of the FLEXIBEAN model, it could necessarily be taken to build up a peer service (Definition 3-10). The client and server composition (modelled by CAT) thereby could serve as the implementation of the remote and local composition part including the internal bindings of component ports (from Definition 3-3). The port bindings between the façade ports (Definition 3-6) could still be established by DCAT.

For the full integration into a service-oriented peer-to-peer architecture, however, some important concepts are missing in the given FLEXIBEAN component model. The composition language CAT does not allow for defining bindings between various distributed applications towards a new more complex application (service composition, Definition 3-17). This is a central concept with respect to the demands of the $SO_{P2P}A$ style. For doing so, the CAT language must be enriched by the API (Definition 3-9) that defines the set of public ports that can be used by another application. Also, it should be possible to define bindings between these API ports.

In order to announce the availability of distributed applications, an adequate advertisement concept (Definition 3-8) must be established. In FREEVOLVE, there is no mechanism for describing applications. A dedicated server acts as the central authority by selecting appropriate applications to its registered users. A user can only select among those applications that have previously been associated to him. New servers and, thus, applications cannot be added dynamically. Both the addition of an advertisement concept as well as the ability to integrate new applications are the ingredients that would shift the FREEVOLVE approach towards a service-oriented architecture.

Although FREEVOLVE already employs ways for user authentication for a controlled, user-related access to applications, more concepts for limiting the accessibility to applications (i.e. peer services) must be conceived. With respect to the presented architectural style in section 4, the notion of self-contained and locatable peer groups (Definition 3-15) are used to regulate the authentication of users to access services. Furthermore, service consumer should fall back on reputation values (section 4.1.2) of service providers in order to estimate the trustworthiness of a provider. Both the peer group and the reputation service are not conceptualized in FREEVOLVE. The notion of server session yet implements a subgroup concept but does not allow for describing and for locating new peer groups. A joining and application procedure (section 3.3.9.2) is also not provided.

The statement of this work is that Stiemerling's proposition of a component model (FLEXIBEAN) and his model for composing applications in a client-server environment (CAT) can necessarily be adopted in parts (i.e. the basic type-based hierarchical port model for composing components towards peer services). For the more demanding requirements, new concepts need to be conceived (i.e. composing of different peer services, advertisement concept, peer groups, and reputation service). The structural model of a distributed application yet serves a solid base to describe a peer service composition but needs to be refined for including the new concepts.

5.3.2 Component-based Tailorability

Component-based tailoring methods as conceived by Stiemerling mainly perform operations based on the addition or removal of ports, parameters, bindings, and components (see [Stiemerling, 2000], p. 138, table 8.1). These methods come up with the adaptation methods as formalized in 3.2.11. In contrast to Stiemerling's methods, the adaptation of parameters is omitted in the $SO_{P2P}A$ style. Commonalities and differences are elaborated in more detail below.

Integration of new Behaviour (Peer Services)

Compared to Stiemerling's tailoring methods, the *discoverPeerService* has been added as an important new adaptation method in the $SO_{P2P}A$ style. This method is used to locate a new peer service in a flexible way. Given a set of found peer services, a suitable service can be determined and eventually bound into a composition (*addBindingAPI* method). This method makes a resulting architecture more flexible and more open towards the supplement of new applications. Further adaptation methods are necessary to adapt service compositions (e.g. *addService*, *deleteService*) as well as further auxiliary methods, e.g. for applying for group membership (see Table 4-2).

Horizontal vs. Vertical Re-Binding Operations

In analogy to the declarative binding of components in CAT, the tailoring methods of Stiemerling perform horizontal re-binding (i.e. binding between ports of various instance components on the same level) and vertical binding (i.e. binding of ports of a component to ports of its parent component). This *shifting* of ports to a higher level is also realized in the $SO_{P2P}A$ style by methods *addPortToFacade* and *addPortToAPI*.

Delegation of Adaptation Operations (Proxy Structure)

In FREEVOLVE, the tailoring methods are applied to the component tree structure residing on the server (Figure 5-3). The component elements delegate the respective tailoring methods to their corresponding proxy parts that reside either on the server or on the client. The proxy elements then execute the tailoring operation directly to the (running) component instances. Note that tailoring operations effected within a single client environment are delegated to all proxy elements and thus client environments. In many application scenarios, this behaviour is rather disadvantageous.

The $SO_{P2P}A$ style demands that only a provider should be allowed to pursue adaptations that eventually are delegated to all depending consumers. Consumers, on the other hand, can affect adaptation methods only in their local environment, that is, they do not influence the appearance of other remote environments. In principle, the proxy approach can be adopted to concretize the similar composition agent (section 3.3.10.4) approach of the proposed architectural style. In order to take the aspect of local vs. global adaptation into consideration, the structural model of a distributed application (Figure 5-4) needs to be adapted adequately. A new proxy type must be developed for maintaining a service composition, that is, bindings among API ports. The location of that proxy thereby depends on the chosen execution model of a service composition (see section 3.3.12). The realization of these models needs to be tackled as well.

Dependency Analysis

In FREEVOLVE, no dependency analysis is realized as proposed in the architectural style of this work (section 4.1.2). The analysis of (subscribed) consumer dependencies

is useful before finalizing the planned adaptation to running instances of a peer service. Without the observance of existing consumer dependencies, functional violations may occur within the local environment of peer consumers. The involvement of the suggested concepts constitutes one of the mandatory contributions to FREEVOLVE, when turning this platform towards a service-oriented peer-to-peer platform.

5.3.3 Deployment of Distributed Applications (Server Session)

In the original work of FREEVOLVE (and, thus, that of Stiemerling), the structural composition plan of a distributed application remains only at the server side. Therefore, the server becomes the only place, where changes to a composition plan can be made persistent. In a peer-to-peer environment, such composition plan must be available and storable at each local environment. This allows each (peer) operator to maintain individual composition plans. At least, a plan describing a high-level composition of different distributed applications (i.e. *service composition*) must be given so that all necessary bindings to depending remote peers (i.e. peer services) can be established at start and managed during runtime (e.g. for exception handling). The local storage of declarative composition plans has not been achieved by the concept of server sessions. Hence, this concept remains inappropriate for the usage in a peer-to-peer environment.

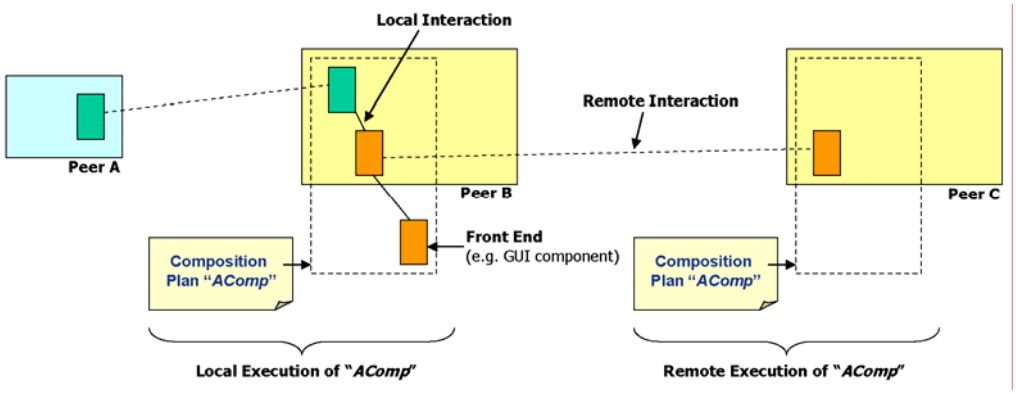


Figure 5-7: From FREEVOLVE towards multi-server environment – first sketch of a possible solution

At first glance, the movement of a FREEVOLVE client towards a peer environment that acts as a client and server of peer services simultaneously becomes straightforward due to the platform's imposed remote capability. Let us assume the existence of a composition plan on a peer B (see Figure 5-7, peer A acts as a provider of a service). Peer B can always interpret this plan to execute the composition consisting of one remote service and one local service. The same plan could also be taken and interpreted by a third-party peer such as peer C. Owing to the given (default) remote capability of a distributed application (service), peer C could also use this composition in the same manner as peer B is able to. To do so, peer C uses the remote front end (this expression will certainly be refined later) which will be executed within C's environment. All other components are instantiated and remain within peer B's environment. For each new request for a peer service, an instance is generated within the provider peer.

Again, Figure 5-7 only displays the first sketch of a peer-to-peer environment without rendering the structural model (including proxies, components etc.) of a distributed level more precisely. The purpose has been to demonstrate the appropriateness of FREEVOLVE as peer-to-peer architecture following the $SO_{P2P}A$ style. The fundamental

requirement for having such peer-to-peer environment is to maintain a composition plan at each network site, the ability to distribute this plan to other peers, and to enable the remote invocation following the principle of Figure 5-7. Apparently, the structural model for a distributed application must be refined thoroughly.

5.3.4 Semantic Integrity Conditions (Exception Handling)

Semantic integrity constraints are used for reducing the risk of deficient tailoring routines in component-based compositions. Integrity checks are carried out during any tailoring step of an end-user. The scope of these checks is the local client environment of an end-user. This way, only local compositions (i.e. the client composition) can be checked to see whether an integrity constraint has been violated.

Integrity conditions as conceived in the $SO_{P2P}A$ architectural style aim at specifying the state of a distributed composition. An integrity constraint serves as a contract among various peers in terms of availability and reliability of services. The adherence of the agreed integrity constraint must be given *during* runtime of a distributed application. This is a principle difference to Won's integrity concepts, where the fulfilment of an integrity constraint is postulated *after* a tailoring step. Hence, appropriate checking algorithms must be available for verifying integrity constraints in regular intervals. According to Won's approach, the underlying checking process is rather lazy, because the integrity is only checked on demand, that is, after a proceeded tailoring step.

Structure of Integrity Constraints

The actual structure of an integrity constraint in $SO_{P2P}A$ (cf. definition 3.10) is similar to the structure found in Won's work. Basically, an integrity constraint comprises a function condition or function and a parameter list (representing the desired state). A process takes these two arguments together with a given composition to see whether a condition is met. A concrete formulation of an integrity constraint in service-oriented peer-to-peer architecture could fall back on the XSemL structure. Apparently, for the special concerns of that type of architecture new conditions must be developed. Also, a context information must be included specifying when an integrity constraint is actually valid (section 4.2.1). An external process has to realize the checking of a condition (e.g. after the detected failure of a peer service).

Exception Detection

The violation of an integrity constraint may not only affect a single peer environment but also potentially many peer environments. The reason for this circumstance is the tendency to have transitive structures as, for example, viewable in Figure 5-7. This leads to further new requirements that need to be faced. Apart from the principle capability to detect the unavailability or failure of peers (see section 3.3.11), adequate techniques need to be developed to forward the event of exception to potentially affected peers. The latter aspect corresponds to the notion of exception cascading (section 4.3.3). For this notion, an adequate protocol needs to be formulated.

Further concepts need to be generated to prevent the exception and, thus, malfunction within a composed application when an integrated peer service is unavailable. In the original structural model of FREEVOLVE, an interaction between a remote and a local part is allowed at any time. There has been no assumption that a server could be unavailable. This assumption, however, is rather weak in a peer-to-peer environment. The $SO_{P2P}A$ style proposes a *broker* condition for avoiding such undesired behaviour

(3.3.11). A broker first ensures that the requesting service is actually available. Therefore, each remote interaction has to be *woven* by a broker. This changes the interaction model of the structural model widely. Besides, concepts for keeping the status of depending consumer peers must be elaborated (cf. *PeerController* process (3.3.11)).

Exception Handling

The $SO_{P2P}A$ style requires that the handling of a violated exception occurs outside the scope of a distributed application. For doing so, separate handler processes should be available (3.2.7) that take over this task. A single handler process consists of a number of actions that can be executed in order to handle the exception. The $SO_{P2P}A$ style also requires that the order of actions is not necessarily fixed but rather *optional* so that an end-user can choose the most appropriate action in respect of a given context. The description of exception handlers could also be in a declarative manner. A given description is transformed to concrete processes at deployment of a composition. Again, XSemL could serve as a starting point for further investigations. For the special demands of a peer-to-peer platform, new action types need to be derived.

The statement of this section is that Won's integrity approach can be adopted partially for implementing the integrity approach of a service-oriented peer-to-peer architecture. His approach must be refined so that an integrity constraint can define the state of an instance of a distributed application during runtime. Further concepts need to be conceived or revised especially for the detection and handling of violated conditions. Apparently, these novel concepts can be adopted for handling simple exceptions, that is, the failure of a single peer service not being part of an integrity constraint. Owing to the tremendous amount of extensions that need to be done in a distributed application (i.e. broker approach, handler, integrity constraints), the structural model needs to be worked over as well to include these aspects.

5.4 Conclusion and next Steps

This section has analyzed state-of-the-art developments around the FREEVOLVE platform with respect to requirements of a service-oriented peer-to-peer architecture. These requirements are mainly imposed by the formal $SO_{P2P}A$ architectural style. Although the principle architectural style of FREEVOLVE is different (client-server), many aspects can seamlessly be adopted for a software engineering realization of a service-oriented peer-to-peer architecture. On the other hand, some concepts need to be revised or are entirely missing. Table 5-1 gives an overview of the required concepts implied by the $SO_{P2P}A$ style (vertical axis) and shows to what extent these concepts can be covered by the state-of-the-art concepts of the FREEVOLVE approach.

One of the main results of this section is that the structural model of a distributed application in FREEVOLVE needs to be completely revised. This is of major relevance to cover the novel and important aspects of the $SO_{P2P}A$ style. In particular, aspects covering the establishment of integrity constraints, exception detection (i.e. broker model), exception handling, and dependency management need to be part of the revised structural model.

Table 5-1 also provides an overview which of the aspired concepts of the $SO_{P2P}A$ style have become part in the DEEVOLVE architecture described within the next three chapters (last vertical column). Each implemented concept is indicated with a section number where more details can be found. DEEVOLVE serves as a reference implemen-

tation of the proposed architectural style of a service-oriented peer-to-peer architecture. Conceptually, it is based on the findings of the FREEVOLVE platform.

	Concept in $SO_{P2P}A$	*Counterpart in FREEVOLVE*	*Appropriateness of Counterpart*	*Changes in Structural Model*	*Overall Effort*	*Implementation in DEEVOLVE*
Component Model	Port Model (Section 3.3.4)	Typed and named ports (event, shared objects)	Is given	No	Low	(Section 6.3.1, Section 5.1.1)
	Component Model (Definition 3-1)	FLEXIBEAN component model	Is given (event primitive must be incorporated)	No	Medium	FLEXIBEAN (Section 6.3.1, Section 5.1.1)
	Component Composition Model (Definition 3-3)	CAT composition model	Is given	No	Low	CAT-XML 6.3.5
Service Model	Peer Service (Section 3.3.7, Definition 3-10)	Distributed Application (DCAT)	Must be extended (advertisement, more types, reputation, peer group)	Yes (Façade, API of Peer Service, peer group, advertisement)	Low	DEEVOLVE peer service (Section 6.3.3)
	Service Composition (Description) (Section 3.3.12, Definition 3-17)	---	---	Yes	High	PeerCAT (Section 6.4.3)
	Service Deployment (Section 3.3.10)	Only simple deployment of distributed application	Must be extended (third-party execution, more execution models)	No	High	Orchestration and Choreographie (Section 6.5)
	Service Discovery (Section 3.3.10.1, 3.3.8.1)	Not implemented		No	Medium	JXTA's discovery concept (Section 6.3.7)
	Advertisement Concept (Definition 3-8, Definition 3-16)	---	---	Yes	Medium	JXTA's advertisement concept (extension) (Section 6.3.6)
Infrastructure	Peer (Section 3.3.8, Definition 3-12)	Server Session Concept	Not appropriate (clients can only host third-party components)	No	Very High	DEEVOLVE peer environment (Section 6.2.1)
	Peer Group (Definition 3-15 Section 3.3.9)	Grouping Mechanism in Server Session	Not appropriate (no advertisement concept)	Yes		JXTA's peer group concept (extension) (Section 6.2.3)
Integrity Concept	Integrity Constraints (formulation)	XSemL, formal presentation	Must be revised for $SO_{P2P}A$ requirements	Yes	Medium	XML-based description in PeerCAT based on XSemL (Section 8.2.2)
	Integrity Constraints (Überprüfung)	Integrity checking Concept	Must be revised (permanent checking, exception cascading)	yes	High	Section 8.3.4
Exception Handling	Broker Concept	---	---	Yes	High	Section 8.3.2
	PeerController (Monitoring, Peer Failure Detection)	---	---	No	high	Section 8.3.1

Exception Handling (continued)	Exception Handling (formulation)	Action parts of solution integrity	Must be revised for $SO_{P2P}A$ requirements	Yes	high	XML-based formulation of handler in PeerCAT (section 8.4.1)
	Exception Handling (process)	Process of handling wrong adaptations	Must be revised for $SO_{P2P}A$ requirements	No	High	Handling flow in DeEvolve(Section 8.4.4), variouls levels of complexity (section 8.5)
	Exception Cascading	---	---	No	Medium	Section 8.3.5
Adaptation	Adaptation Methods (Section 4.1.1)	Tailoring strategies	Must be extended w.r.t $SO_{P2P}A$ requirements	No	Medium	For handling exceptions (section 8.4.2), TailoringAPI (section 5.3.2)
	Tailoring published service (Provider) (Section 4.1.3)	Tailoring Environment	Should be extended w.r.t. new concepts (e.g. API, Façade).	Yes	Medium	Section 7.5
	Tailoring Interface of service (Consumer) (Section 4.1.4)	Tailoring Environment	Only proxy structures should be adapted, no delegation allowed	Yes	Medium	Section 7.5
	Composition Agent (Delegation) (Section 3.3.10.4, Section 4.1)	Proxy structure	Is given (hierarchical structure), further proxy type for maintaining service compositions	Yes	Medium	Section 6.3.4, Section 4.1
Dependency Management	Adaptation Policy (Section 4.1.2, Section 3.3.9.1)	---	---	Yes	Medium	Section 7.3
	Consumer Subscription (Section 3.3.10.6)	---	---	No	Medium	Section 7.2
	Dependency Value (Subscription) (Section 3.3.10.6)	---	---	Yes	Medium	Section 7.2.3
	Dependency Analysis (Process) (Section 4.1.2)	---	---	No	High	Section 7.3.3 Section 7.4.2
	Reputation value (assessment) (Section 4.1.2)	---	---	Yes	High	*Not implemented*

Table 5-1: Overview of the concepts of the $SO_{P2P}A$ style and their counterparts in FREEVOLVE

What one can also see from this table is that FREEVOLVE has hardly addressed the concepts of exception handling and dependency management. These concepts will therefore be the focus of the chapters 7 and 8. At first, the basic concepts of DEEVOLVE will be elaborated in chapter 6.

Chapter 6
The DEEVOLVE Runtime Environment

This chapter presents fundamental aspects of the DEEVOLVE runtime environment. DEEVOLVE is the default implementation of the proposed *distributed runtime environment* of the $SO_{P2P}A$ architectural style. DEEVOLVE allows for deploying and maintaining peer services that conform to $SO_{P2P}A$'s formalized model of peer service. This chapter addresses aspects referring to the structural organization of DEEVOLVE, its underlying component-based service model (including service composition and execution). Besides the illustration of those conceptual issues, the chapter gives a summary of both the prototypical implementation and preliminary evaluation results. Structural and behavioral aspects of the service model and the architecture are modeled in a semi-formal way by means of extended UML diagrams. Further more specific aspects of DEEVOLVE will be illustrated in section 7 (adaptation strategies, dependency management) and section 8 (exception detection and handling).

6.1 Overview of the Architecture

This section aims at providing a general overview of the DEEVOLVE runtime environment. In particular, it outlines the relation of DEEVOLVE to the FREEVOLVE (chapter 5) architecture and to Sun's JXTA peer-to-peer framework [Sun, 2005a].

6.1.1 Relation to FREEVOLVE

Chapter 5 has highlighted the close relationship between the desired properties of the $SO_{P2P}A$ architectural style and the conceptual model of the FREEVOLVE platform. This work therefore adopts the conceptual model of FREEVOLVE for implementing DeEvolve at least to some degree (cf. Table 5-1). The following issues roughly summarize the prospects for adopting FREEVOLVE as a fundamental architecture:

- The availability of a port-based component model (FLEXIBEAN) that enables remote interaction between distributed components
- The orientation of FREEVOLVE towards an architecture allowing for *tailoring* component-based composition
- The orientation of FREEVOLVE towards a distributed component-based software architecture
- The availability of a sound (and proven) Java implementation as well as sophisticated tools.

In addition, a couple of useful groupware applications have already been available that turned out to be useful also for the context of DEEVOLVE and its supporting application scenarios (cf. section 9.5). Keeping the compatibility of FREEVOLVE applications within the DEEVOLVE environment has therefore become an important requirement.

In fact, many concepts of $SO_{P2P}A$ are not covered by FREEVOLVE (see Table 5-1). Additional frameworks have been analyzed in particular for properly realizing the proposed service model of $SO_{P2P}A$. In order to meet recent standards for the development of peer-to-peer architectures, the JXTA framework has been chosen as a promising solution. The relationship of JXTA to DEEVOLVE is elaborated in the next section.

6.1.2 JXTA as the fundamental Framework

The JXTA framework has been the second major framework that has been chosen for the conceptualization and implementation of DEEVOLVE. Relevant concepts of this framework have been adopted. Most notably, the platform uses basic protocols and standards of the service model by JXTA for realizing its fundamental service model. JXTA also provides the fundament for implementing the peer group concept, for having a general-purpose communication channel (PBP protocol), and the advertisement concept. For booting a peer environment, the configuration process has been taken.

The reason for taking JXTA can be justified mainly by two reasons. Firstly, as already mentioned in 6.1.1, JXTA constitutes – at the time of writing this thesis - the most sophisticated framework for developing peer-to-peer architectures. The underlying concepts are proven and well-established. The second reason is that JXTA comes with a sound Java-based reference implementation with a good documentation.

6.2 Peer and Peer Groups

The principle structure of the DEEVOLVE architecture is displayed as a *structural model* in terms of a UML class diagram (Figure 6-1).

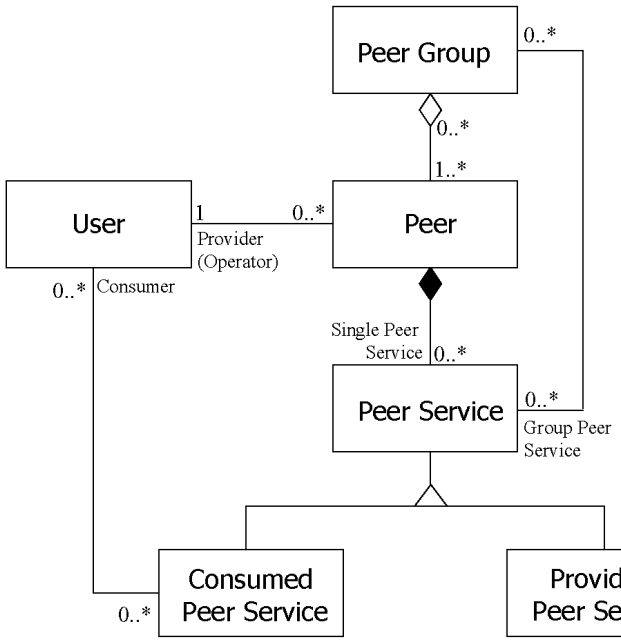


Figure 6-1: The structural model of the DEEVOLVE infrastructure – UML class diagram

The purpose of a structural model is to model *roughly* the important architectural elements of DEEVOLVE and to depict their dependencies. The next subsections explain each element of the model depicted in Figure 6-1 in more depth.

6.2.1 Peer

The central concept of the structural model is a peer (class Peer). A peer (hereafter also referred to as a DEEVOLVE peer) represents a *networked runtime environment* in which a user is able to provide peer services to other peers and, at the same time, to consume peer services from other third party-peers. In this context, the user adopts the role of a provider. A peer is always associated to a user who acts as the provider (or operator) of that peer. Potentially, a user run several peers in parallel. Although a peer is often regarded as an arbitrary network-device (e.g. PDA, telephone, mobile device etc.), it is assumed that a peer is a normal personal computer device (i.e. personal computer or laptop) being connected to the Internet (e.g. via DSL, (W)LAN, UMTS enabled Internet access points).

A peer can uniquely be identified within a network by an address. For utilizing this concept, DEEVOLVE adopts the identifier concept of JXTA ([Sun, 2005b], p. 14). A JXTA ID serves as a canonical way of referring to that peer. Besides referring peers, the identifier concept is also used to locate other entities such as peer groups and services. For expressing identifiers, JXTA uses URNs. URNs are presented in a textual form. An example for a URN serving as a unique peer ID is:

urn:jxta:uuid-ABE464FE12DEA42ACF68415EABB12DE3

JXTA offers a Java-based service for randomly producing such URNs. More information can be read in JXTA's specification or user guide [Sun, 2005b].

6.2.2 Peer-to-peer architecture

A peer-to-peer architecture conforms to the collection of all available, potentially interconnected peers (see Figure 6-2). Each local peer environment is thereby part of a distributed (DEEVOLVE) runtime environment. DeEvolve supports two types of peers, *simple peer* (or just peer) and *rendezvous peer*.

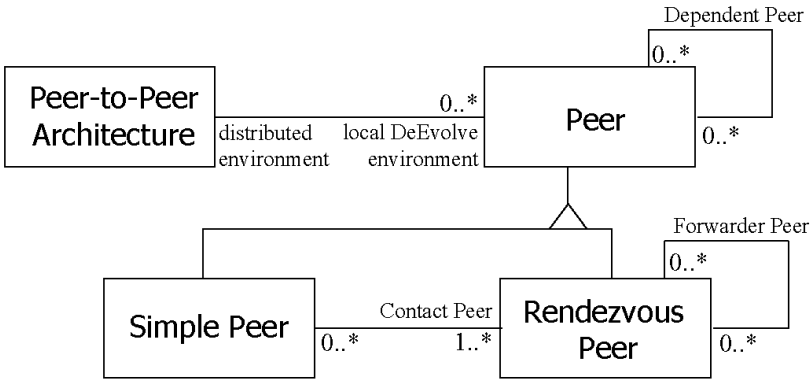


Figure 6-2: The types of a peer. A simple peer must always be connected to rendezvous peer for route out advertisements

A rendezvous peer takes over responsibilities to distribute advertisement within a peer-to-peer architecture. A simple peer always has to be connected to at least one

rendezvous peer. Any new peer service described by an advertisement has to be published to at least a subset of known rendezvous. The rendezvous peers then disseminate these advertisements to other rendezvous peers according to a pre-defined crawling algorithm and a global DHT data structure (see section 2.2.2 for more details). Any rendezvous peers caches advertisements. A simple peer also can send queries to the rendezvous peer for locating specific advertisements. A rendezvous peer is then asked to forward the query as long as a rendezvous finds matching advertisement in its cache. All discovered advertisements are sent back to the originating requester.

In practice, a rendezvous peer represents a highly available, reliable, and trustable peer. In typical project settings (see scenarios in construction informatics in section 9.5) these peers correspond to project leaders or any other kind of responsibility. Aspects concerning the efficiency of routing and crawling algorithms, suitable global data structures and scalability issues are beyond the scope of this work.

6.2.3 Peer Group

A *peer group* is a collection of different peers that agreed upon a common understanding, interest, knowledge, competency, or simply a set of services. A peer may potentially belong to many peer groups simultaneously. In addition, a peer service can be assigned one or more peer groups. A peer willing to use a peer service must then be a member of all associated peer groups. The peer group concept is realized in the style of the group concept provided by the JXTA framework. The association of peer services to peer groups is not supported in this framework. JXTA has been extended at that point accordingly (see [Alda and Cremers, 2005] for more details).

JXTA also provides a protocol (Group Membership Protocol, GMP) enabling peers to apply for and join a peer group. Each peer group can thereby dictate its own membership policy from open (anybody can join) to restricted (password enquiry or personal acknowledge required). The concept of the membership protocol and how it is used in DEEVOLVE is not further described in this work.

A peer group can determine a set of equal peer services provided by its constituting peers as a *group peer service*. The redundancy of peer services within a group increases the availability of a peer service. If a single peer fails, the collective peer service is not affected as the same instance of service is still available from another provider peer. A peer group service assumes that the peer service is stateless, that is, it does not maintain a state across many service requests. An applicable example would be a printer service. A group peer service is made available to other peers as part of a peer group advertisement (see below for more information).

By default, each DEEVOLVE peer environment is a member of the global "DeEvolve" peer group. The "DeEvolve" group provides a set of *basic group services* that each peer needs for discovering and publishing resources within a peer-to-peer architecture. Also, basic group services are available for joining and applying to a peer group. These services belong to the reference implementation provided by JXTA and have been adopted for DEEVOLVE. Unlike regular peer services described by a CAT-XML (6.3.5), group services cannot be composed with other services. Basic group services can be used through a Java-API implementing JXTA's core protocols for discovering and publishing services (PDP) and realizing the group membership service.

If a DEEVOLVE peer boots for the first time, the environment tries to locate the peer group advertisement of group "DeEvolve" and eventually joins that group. Given

the case that the “DeEvolve” peer group has not been created yet, the first peer creates a new peer group “DeEvolve” and advertises its existence within the “NetPeerGroup”. This peer group is a global peer group provided by Sun that acts as an anchor of all new peers willing to create their own peer group(s).

6.2.4 Advertisements

All kind of resources supported in DEEVOLVE – peer services, peers, and peer groups – can be published, that is, announced by advertisements. An advertisement is a language-neutral metadata structure represented as a XML document. The basic structure of an advertisement is adopted by JXTA. Also, JXTA’s protocols for publishing and discovering an advertisement are used. An advertisement is augmented by a lifetime attribute (TTL). This attribute specify the validity of an advertisement. It is possible that a single resource can be described by many (old and new) advertisements at the same time. A peer operator retrieving both old and new advertisements of the same peer service then has to decide, which of these are more appropriate.

Advertisements are not pushed to a central directory or index (cf. a UDDI directory in a Web service architecture), but assimilated within a peer-to-peer architecture. Also, queries to locate advertisements are directed to the peer-to-peer architecture. More information on both locating and publishing advertisements are elaborated in section 2.2.2. The advertisement of the DEEVOLVE peer group is depicted in Figure 6-3.

```
<?xml version="1.0"?> <!DOCTYPE jxta:PGA>
<jxta: PGA xmlns:jxta="http://jxta.org>
   <GID> urn:jxta:jxta-DeEvolveGroup</GID>
   <Name> DeEvolveGroup </Name>
   <Desc> This is the entry group for a DeEvolve peer </Desc>
   <GroupServices>
      <Service>
        <Name> MailService </Name>
        <MSID> urn:jxta:uuid-DEABD4D4637E363AEB4B4…</MSID>
      </Service>
   </GroupServices>
   <AdaptationPolicy>
      <Strategy> Notification </ Strategy >
      ....
   <AdaptationPolicy>
</jxta:PGA>
```

Figure 6-3: The structure of a peer group advertisement (example of the DEEVOLVE peer group advertisement including one group peer service)

The main header of this advertisement describes the name of that group, an informal description and a collection of applicable peer group services. Each entry of the collection <GroupServices> describes a group service indicating the name and the pertaining peer service specification. Within the <AdaptationPolicy> tags, the adaptation policy for regulating service adaptations within that group are specified (see chapter 7 for more information). Advertisements for peer services are described in section 6.3.6.

6.2.5 Modeling Peers and Peer Groups for concrete Applications

This work uses parts of the UML 2.0 modeling language [Booch *et al.*, 2005] for accurately modeling the architectural elements of DEEVOLVE for a concrete application. In

contrast to the structural model, whose purpose it is to model the general structure of DEEVOLVE, these models are used to model concrete applications that can be deployed within DEEVOLVE. So, these *application models* do not possess cardinalities for expressing the relationships between elements. This notation should also be the base for future CASE tools accomplishing developers and users to create or to adapt compositions (see 6.6.5). Some UML elements have been introduced in chapter 3 for clarifying the formalisms of the $SO_{P2P}A$ style. In the following, the UML-based notation will be refined with respect to the characteristics of the DEEVOLVE runtime environment. Since UML does not provide general elements neither for peer-to-peer nor for service-oriented architectures, adequate extensions will be proposed.

For representing peers and peer groups, deployment diagrams are applied. According to Booch et al., a deployment diagram is a diagram that shows the configuration of run time processing nodes and the artifacts (in this work: peer services) that live on them. An overview of the core elements is depicted by means of a small example in Figure 6-4. A DEEVOLVE peer environment is represented by a node. A node is indicated by a name and by a *stereotype* indicating the type of a peer (<<peer>> for a normal peer or <<rendezvous peer>> for a rendezvous peer). By using a stereotype, it is possible to extend the core language of UML with new elements. Furthermore, a peer lists all its provided (and published) peer services as well as all peer services it consumes from other peers. A consumed service can be prefixed by the name of the peer provider. Peer Services can be further detailed by composite structural diagrams (see Figure 6-16) that can be placed directly below the peer node (see section 6.5.2 for examples).

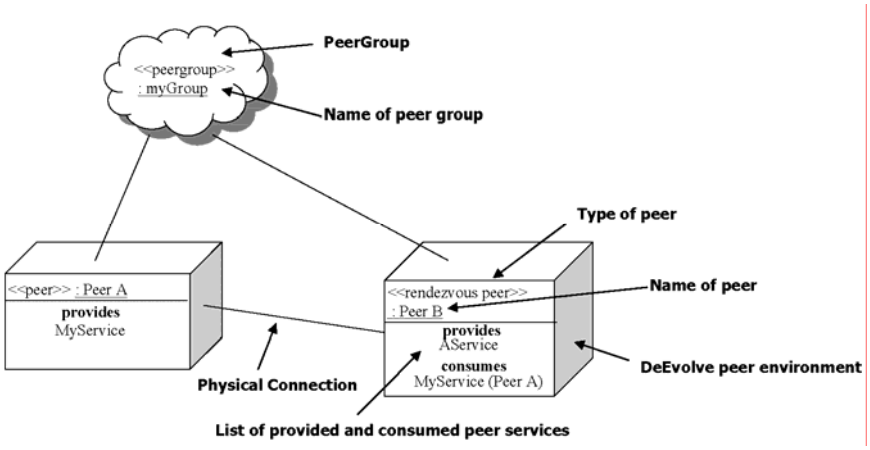


Figure 6-4: UML-based notation for modeling peers and peer groups

There can be *physical connections* between peers denoting a dependency relation. A connection could, for instance, be an Ethernet connection. A peer group is visualized by a cloud. A cloud comprises a name of a peer group. The Stereotype <<peergroup>> indicates the purpose of this node. The membership of a peer to a cloud is indicated by a connection. The example of Figure 6-4 consists of two peers (“Peer A” and “Peer B”) each associated to peer group “myGroup”. Peer “Peer B” consumes the peer service “MyService” provided by Peer A. It also provides service “AService”.

6.3 Peer Services

This section elaborates basic concepts of the service model of DEEVOLVE. At first, the underlying component model is introduced first. It is then shown, how peer services can be modeled by components.

6.3.1 Component Model

With respect to the requirements of the $SO_{P2P}A$ style, peer services are modeled internally by components. To realize a component model for composing peer services, DEEVOLVE adopts the FLEXIBEANS component model. At the lowest level of this component model, so-called *instance components* ("iComponents") correspond to concrete behavior that in turn commensurates with Java classes. Compared to the original notion of Stiemerling, "iComponents" are modeled in a UML-based notation adopting the graphical model of a component diagram (Figure 6-5). A component consists of a number of ports that are further described by a port type (i.e. a Java interface), a unique name, and a polarity (required or provided). Following the FLEXIBEANS model, two concrete interaction primitives are supported, event notification and shared objects (indicated by extensions "_event" and "_shared" at the end of the type qualifier). The shared object primitive serves as the direct implementation of the port model as recommended in the architectural style. The $SO_{P2P}A$ style also allows for simulating the event notification pattern (see section 5.3.1). In contrast to $SO_{P2P}A$, the polarities for event notification are interchanged. The provided port (filled circle) denotes an event producer, not an interface provider.

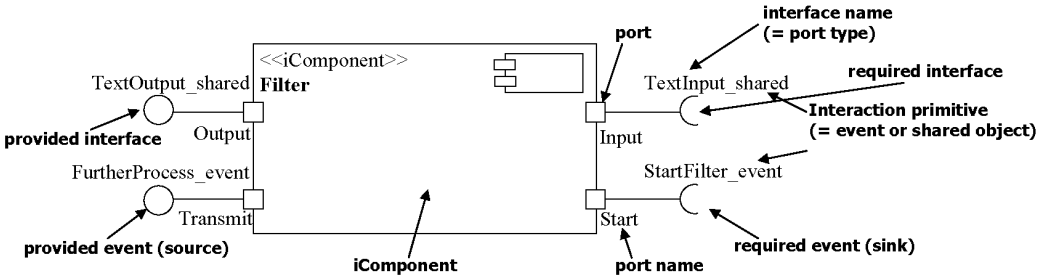


Figure 6-5: Model of a FLEXIBEAN instance component ("iComponent").

In face of the inclusion of the event notification primitive and its corresponding polarity change, the FLEXIBEAN model still satisfies the component definition of the $SO_{P2P}A$ style (Definition 3-1). The operations of the *Reflectionprocess* are not modeled explicitly. The operations for adding and removing port references must be provided in a FLEXIBEAN component definition (see [Stiemerling, 2000], section 7 for an overview of the signatures). Operations for adding and removing ports are provided by the reflection capabilities of a Java class (see [Flanagan, 2005], pp. 283 ff).

The example in Figure 6-5 shows a single component implementing a filter algorithm. This component can filter a given text (provided by shared object "TextInput" along the "Input" port) according to an internal filter strategy (e.g. stripping out all blanks). An external component can start the process of filerting by sending an event to the port "Start". After the filtering process, the component is capable of transmitting the revised text to some external component through ports "Transmit" and "Output".

6.3.2 Component Composition

Concrete instance components can be composed to higher-level compositions. The model of a composition is presented in Figure 6-6. This model is based on a composite structure diagram provided by UML.

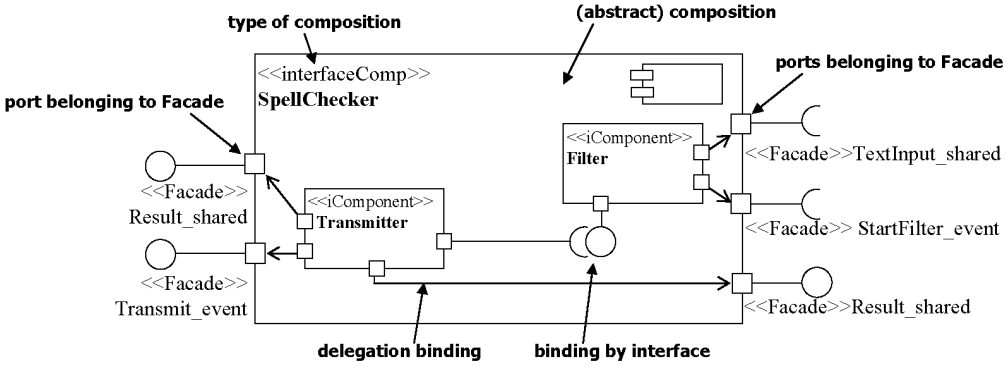


Figure 6-6: The model of a FLEXIBEAN composition aggregating various iComponents together. The example shows a spell checker consisting of two iComponents.

A composition element is similar to the abstract composition concept given by Stiemerling. In fact, the term "abstract" is misleading, because it might denote an artifact that cannot be instantiated (cf. definition of abstract object in [Bruegge and Dutoit, 2004]). To avoid conflicts, the term composition has been chosen here. A composition can be assigned a composition type (indicated by a stereotype). Possible composition types are "interfaceComp" or "localComp". Either type is used for declaring the interface composition part and the local composition part of a peer service, respectively. With respect to $SO_{P2P}A$ (Definition 3-5), ports of iComponents can be shifted or delegated to the façade of a composition. Some of the façade ports can also be used to define the API of a peer service. Communication between two compositions can only take place through the ports of the respective compositions' facades. This model resembles the formal model of component composition (Definition 3-3) in $SO_{P2P}A$. The *binding* ports describing the concrete bindings between ports are described by using the CAT-XML language (section 6.3.5). This way, these bindings are not directly a part of the composition, but are maintained in a separate file.

In the example of Figure 6-6, a composition is shown consisting of the filter component of Figure 6-5 that is connected with a transmitter component. The ports of the filter component have been defined as the façade ports. The ports of transmitter component allows for conveying the revised text and the flow of control outside the composition. The revised text can be obtained through the shared object of port "Result".

6.3.3 Peer Service Model

A peer service is composed out of two compositions namely that of a local composition part and that of an interface composition part (Figure 6-7). The latter composition represents the interface provided by a peer service. For remote usage of a peer service, this interface composition needs to be migrated to the requesting peer. The same interface composition is also used for local usage of a peer service. This is possible due to the transparent remote interaction facility of the FLEXIBEAN component model. The remote interaction between interface and local composition is carried out between the façade ports of these compositions. Ports representing the public interface or API of a

peer service (marked by stereotype <<API>>) are delegated from the façade ports. API ports can belong to either composition type.

A peer service is surrounded by a box indicating its boundary. A peer service is augmented by a name and a peer service type (stereotype <<peer service(type)>>, where "type" can be "shared" or not "not_shared". "Shared" means that the local composition part is shared between many consumers. Attribute "not_shared" indicates that for each new peer requesting a service a new local composition part is instantiated and, then, reserved for that consumer (see section 6.5.3 for more details).

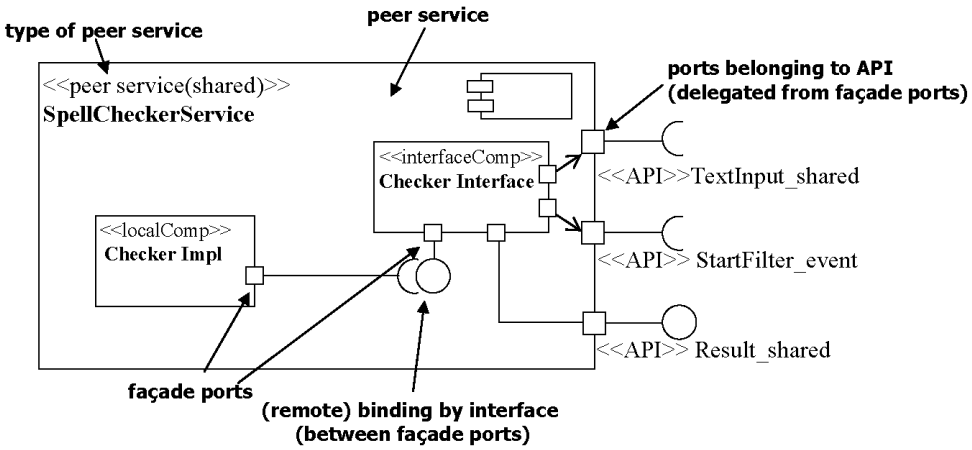


Figure 6-7: The model of a peer service composed out of two compositions. The example depicts a peer service for checking the spelling of a text.

For the sake of readability, the box representing the peer service can be omitted. The names of the composition parts should then reference the corresponding name of the peer service (e.g. "SpellCheckerService(LocalComp)"). The API ports need to be associated to the respective façade ports. This variant is reasonable when describing the distribution and deployment of peer services across many peers (see Figure 6-16).

The peer service depicted in Figure 6-7 consists of the composition illustrated in Figure 6-6 acting as the interface part. The service implements a spell checker service. The interface takes the text, filters out unnecessary characters and sends it to the local part along the remote bind for performing the actual spelling check. Subsequently, the final document including remarks can be obtained through the shared object of the "Result" port. The necessary API ports for interacting with the peer services are represented by the facade ports of the interface composition.

6.3.4 Structural Model of a Peer Service

The structural model of a peer service is based on the structural model of a distributed application within the FREEVOLVE architecture (section 5.1.3). It shows the relationship among components, peer services, and the proxy structure, which is necessary to maintain (i.e. to deploy and adapt) individual components. For meeting the requirements of the $SO_{P2P}A$ style, a couple of refinements are suggested.

Principle Structure

The central concept of the structural model (see Figure 6-8) is the "Peer Service" class. This class encapsulates the "DistComponent" class representing a distributed application. An instance of class "DistComponent" takes in either the role of a 'Re-

motePart' or that of an 'InterfacePart' (see corresponding role annotations at the class association relation of "DistComponent"). The interface part represents the interface composition part of a peer service that resides in the peer environment of a consumer. The local part represents the local part composition of a peer service that remains in the peer environment of a provider

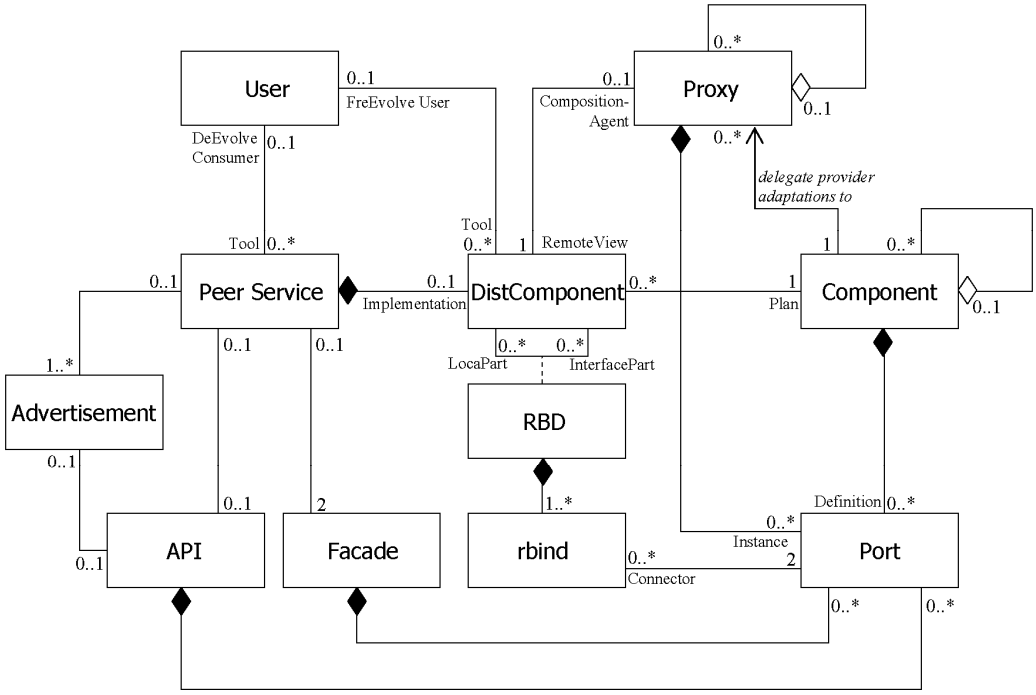


Figure 6-8: The structural model of a peer service in DEEVOLVE – UML class diagram

Holding the role 'RemotePart', an object of class "DistComponent" denotes the remote view on a peer service. It then indirectly references a "User" object (holding the role 'DeEvolveConsumer') through a "Peer Service" object. Class "User" indicates the subscribed peer user that has bound the peer service. This process is dynamic since users can bind a service arbitrary during runtime. This is in contrast to FREEVOLVE, where a *static* list of users is given at start time. Besides, a "DistComponent" object points to the proxy structure that represents the instantiated remote components of the interface composition. The proxy structure is a concretization of the composition agent concept (section 3.3.10). In the structural model, this is adumbrated by the role 'CompositionAgent' at the association between "DistComponent" and "Proxy".

Possessing the role of a 'LocalPart', an object of class "DistComponent" references the component tree structure ('C' elements) that corresponds to the structural plan of the distributed application. It also references to the local proxy structure representing the instantiated components of the local composition. Note again that the proxy structure is an exact copy of the component structure. A component structure can hold potentially many copies. Each adaptation to the component structure is delegated to the corresponding proxy structure, which applies these to the corresponding abstract components or to the concrete component instances[19]. In contrast to FREEVOLVE, however, the reversed direction is not supplied: adaptations to the proxy structures are not delegated to the pertaining component structure. The uni-directional association between

[19] The adaptation to the component structure presumes the analysis of consumer dependencies. This aspect is omitted now for the sake of clarity and brevity.

class Component and class Proxy shows this. Therefore, adaptations to local proxy structures only have local effects. This approach is in accordance to the adaptation model of the $SO_{P2P}A$ style (see 4.1).

Façade and API

Class Peer Service also references two new classes: “Façade” and “API”. These classes realize the respective notions of the façade of a peer service (Definition 3-5) and the API of a peer service (Definition 3-9). An object of class Façade represents the interfaces of the local and interface compositions of a given peer service. Façade ports can be applied for remote interaction between the local and the interface composition parts. A peer service references exactly two facades. A façade is a composition of an arbitrary number of ports that belong to the constituting components of a distributed application and thus of a service.

An object of class API represents the public interface of a peer service. A peer service may provide at most one API but can also omit it (cardinality ‘0..1’). In the latter case, the peer service is a closed application. An API is useful when composing many different peer services towards a service composition (see 6.4). Peer service composition can only occur through the ports of an API.

Compatibility to FREEVOLVE

A user is still enabled to use a distributed application of the original FREEVOLVE platform without describing, publishing, and locating it as a peer service. A peer can therefore address a pre-defined FREEVOLVE server in which it has been registered. The (IP) address of that server can be entered in a setting file. In the structural model, this is indicated by the two roles ‘FreEvolveUser’ and ‘DeEvolveConsumer’ a consumer may take in (class User). Being a ‘FreEvolveUser’, a user is able to access directly a distributed application without falling back on the peer service concepts. Holding the role ‘DeEvolveConsumer’, he is able to utilize the complete feature of the service model of DEEVOLVE.

6.3.5 Modeling of Peer Services with CAT-XML

For the declarative modeling of a (component-based) peer service, the CAT language has been adopted. However, the CAT language itself exhibits some weaknesses. Its syntax is proprietary and not based on standards (i.e. XML). Furthermore, it is based on the concepts of aggregating components towards abstract components, which is - as it has been mentioned earlier - a bit confusing if not wrong. For the purposes of this work a new CAT variant (CAT-XML) has been designed based on the meta-language XML. For the deployment of a composition into DEEVOLVE, a compiler is needed to compile a CAT-XML file into a CAT file. The reason for this approach is justified by the implied requirement of DEEVOLVE to remain compatible with conventional FREEVOLVE-based applications.

CAT-XML is splitted into two parts, one for defining the constituting compositions (local and interface), the other for defining the peer service. An example for a composition declared with CAT-XML is depicted in Figure 6-9. A CAT-XML declaration of a composition essentially consists of five parts (each annotated by a brace). At first, the ports belonging to the façade of the composition are defined. Each port is described by its polarity, its type (refers to a Java class) and a unique identifier.

Figure 6-9: An example for a CAT-XML based composition (excerpt)

By convention, an event type ends with "_event", a shared object type with "_shared". Next, the instance components ("iComponent") for this composition are declared. In the third part, concrete subcomponents of iComponents are defined. For each iComponent many subcomponents can be defined. Then, concrete horizontal bindings between the ports of the defined *subcomponents* are established. Finally, vertical bindings from subcomponents to façade ports are defined. These latter bindings realize the delegation of ports to higher-level façade ports. Figure 6-9 illustrates the CAT structure for the interface composition depicted in Figure 6-7.

Finally, a peer service is declared by a separate part that aggregates two compositions to a peer service composition (Figure 6-10). Each composition is assigned a dedicated type stating whether it is a local or interface composition part. Besides, the API of that peer service is defined. In the lower part, the remote bindings between local and interface compositions as well as the vertical bindings from façade to API ports are specified. The example in Figure 6-10 shows the CAT definition of the peer service described in Figure 6-7.

6.3.6 Advertisement of Peer Services

A peer service can be described by one-to-many advertisements. The operator of a peer environment is capable of locating such advertisements either via the standard console of JXTA or through the DEEVOLVE console (see section 6.6.3). He then is able to decide whether a peer service meets his expected requirements.

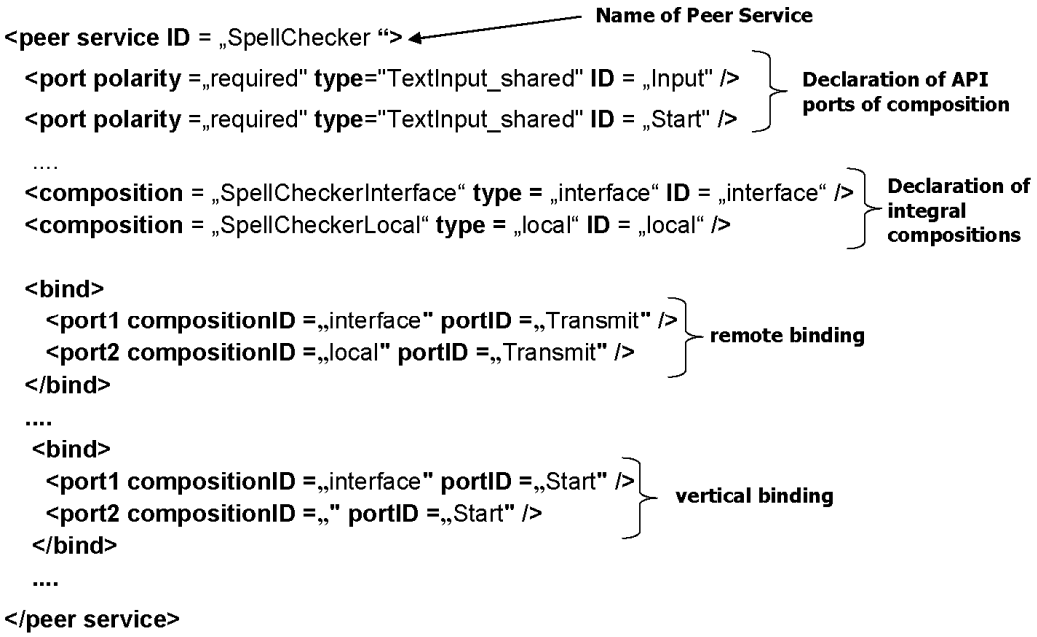


Figure 6-10: An example for a CAT-XML-based peer service (excerpt)

Three types of advertisements, service class advertisement, service specification advertisement, and service implementation advertisement describe peer services:

- A *service class advertisement* is used to advertise the existence of a peer service. It merely contains of a service class ID (unique identifier to reference the peer service), a representative name and a description. This type of advertisement also announces the peer group memberships to which a dedicated peer service belongs.
- A *service specification advertisement* contains the specification of a service. The specification comprises all information used to access a peer service. For a given class advertisement, there can exist many different specifications (e.g. for different platforms or for different use cases). A service specification advertisement always reference to its corresponding class advertisement. Apart from many auxiliary tags (amongst other, a name, a description, and a UUID), a service advertisement includes the public interface (API) of a peer service. This API is separated by the CAT-XML description of a peer service.
- The *service implementation advertisement* provides information on the implementation of a peer service. This advertisement type is relevant for executing composed peer services (cf. section 6.5). In this case, it incorporates the PeerCAT description file. This file also provides a default exception handler that can be used by a local peer operator for realizing exception handling on this peer service. An implementation advertisement always points to a specification advertisement.

More information on the advertisement concept can be found in [Mitrov, 2003]. The DEEVOLVE prototype provides tools for generating advertisements (section 6.6.5).

6.3.7 Peer Service Discovery and Publication

DEEVOLVE uses the discovery service of JXTA to realize the discovery and publication of peer services. Apparently, no peer services are discovered but their corresponding advertisements. More information on the concept beyond discovering and publishing of advertisement can be found in [Sun, 2005a].

6.4 Peer Service Composition

The particular strength of DEEVOLVE lies in its advanced composition approach. Peer services provided by various peer providers can be bound into a local environment. Moreover, these services can extend local compositions. With respect to the core idea of $SO_{P2P}A$, a service composition can not only be consumed by a local consumer but also by third-party consumers. In this section, the first sketch of a service composition model (section 5.3.3, Figure 5-7) will be refined circumstantially.

6.4.1 Modeling a Peer Service Composition

A service composition is a composition of many local or remote peer services. The composition of peer services results from a number of bindings between ports of the pertaining APIs of these services. Figure 6-11 shows a service composition represented by a UML composite structure diagram. This service composition consists of two peer services, "WordProcessor" and "SpellChecker". The purpose of this composition is that a simple word processor can use the service of a "SpellChecker" for checking the spell of a text. The surrounding boxes of the peer services have been omitted to improve the readability of the overall diagram. The surrounding box of the service composition contains only of those composition parts, where concrete service bindings have been established. All other parts may placed outside the box. A service composition comprises a name and a stereotype (<<service composition (type)>> for indicating the type of a service composition. Type is either "local" or "published". If type is "local", the service composition is only used internally. Otherwise, if the type is "published", the service composition is advertised as a *composite peer service* that can be found and used by third-party peers.

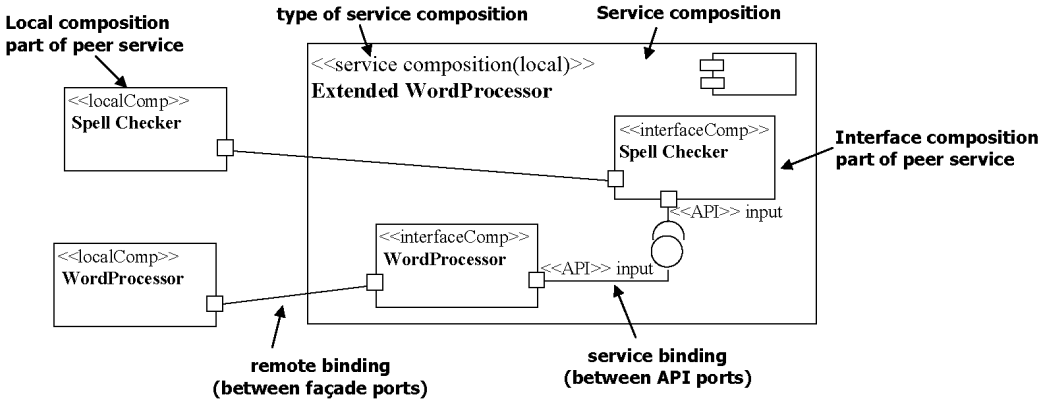


Figure 6-11: Model of a service composition aggregating two peer services. The example pictures a service composition yielding an extended word processor

6.4.2 Structural Model of Service Composition

The structural model of a service composition (see Figure 6-12) is derived from the structural model of a (single) peer service (see Figure 6-8). In this figure, only relevant concepts have been taken into consideration.

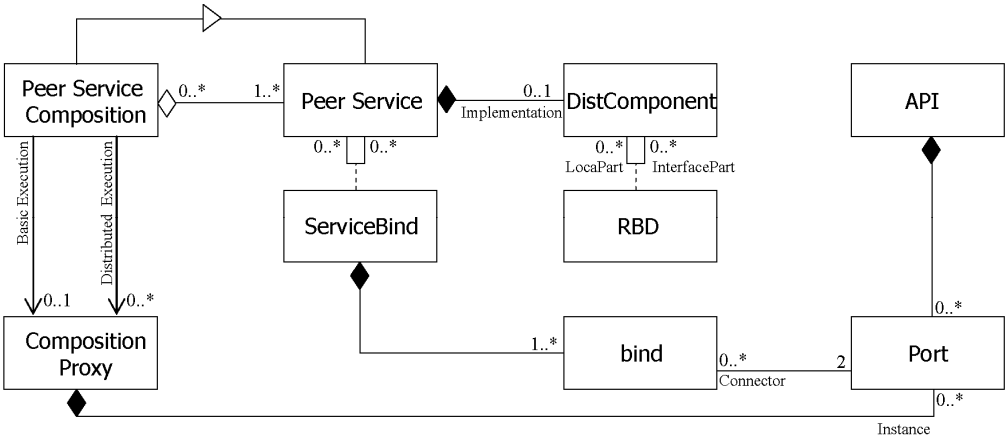


Figure 6-12: The structural model of a peer service composition.

A service composition is an aggregation of at least one peer service. A concrete service composition is constituted by a number of bindings among ports that belong to the respective public interfaces (class “API”) of the participating peer services. Note that these ports must belong to the API of a peer service (see composition association between class “Port” and “API”). In principle, there is no distinction between a local or a remote peer service, that is, a composition can consist of services provided by third-party peers as well as local (potentially unpublished) peer services. In fact only (DEEVOLVE-compatible) peer services featuring an API can serve as a candidate service for a service composition. This means also that a distributed FREEVOLVE application cannot be composed directly with other peer services. It must first be changed to a peer service. A service composition can itself be a peer service again that can be published in the peer-to-peer architecture (indicated by the inheritance relation).

Class “PeerServiceComposition” represents the plan of a service composition and only *implements* those adaptation methods suggested by $SO_{P2P}A$ (cf. section 4.1.1) that are applied on the level of the service composition only (e.g. addService, deleteBindingOfAPIPorts). It can also *delegate* adaptations to *internal* components to the respective ‘C’ structures. An object from that class can reference objects of class “CompositionProxy” that represent a service composition during *runtime*. Depending on the chosen execution model (see section 6.5), a service composition can refer to one (basic execution) or many (distributed execution) proxies. A “PeerServiceComposition” object uses its associated proxies to delegate the affected adaptation methods to all running instances. In analogy to the original component/proxy structure of FREEVOLVE, class “PeerServiceComposition” stores all changes persistently in the corresponding, local PeerCAT file.

The collection of available adaptation methods are placed in the “Tailoring” service (see section 6.6.1). More information on the interactions among that service, the plan structures, and the runtime proxies, and the services using the Tailoring can be found in section 8.4.4.

6.4.3 PeerCAT Composition Language

The PeerCAT composition language has been developed in order to accomplish the declarative composition of peer services towards more complex and meaningful service compositions. PeerCAT is completely based on XML. The complete syntax of PeerCAT is defined by a document type definition (DTD) (see [Palij, 2006]).

Figure 6-13 shows the principle structure of a PeerCAT file. It consists of five parts. The first part summarizes all peer services that are part of a given composition. The second part defines the bindings between the API ports of these peer services. The third part consists of declarative exception handlers. These handlers are used to describe procedures in the case when single peer services become unavailable. Integrity constraints for the given service composition can be entailed in the fourth part. Handlers for defining procedures when these integrity constraints become violated can be described within the third part. The fifth part states dependency values for each remote peer services indicating the local relevance for this service.

```
<composition name = “aName" ID = “anID“ >
   <services> …. </services>
   <bindings> …. </bindings>
   <exceptionHandling> …. </exceptionHandling>
   <semantics> …. </semantics>
   <dependencies> …. </dependencies>
</composition>
```

Figure 6-13: Structure of a PeerCAT file

This section concentrates on the introduction of the first two parts, <services> and <bindings>. PeerCAT structures for exception handling will be elaborated in section 8.4.1. The dependencies part indicating the relevance of dependent peer services will be introduced in section 7.2.2 (consumer dependency analysis).

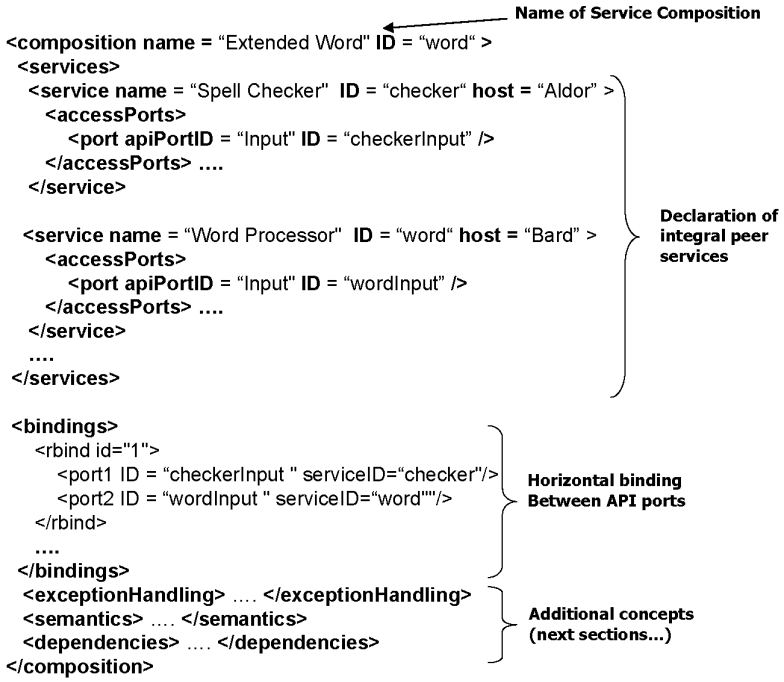


Figure 6-14: Example for a service composition with PeerCAT

An example for a PeerCAT service composition is depicted in Figure 6-14. This example corresponds to the graphical service composition illustrated in Figure 6-11. After the indication of the constituting peer services (“SpellChecker” and “WordProcessor”), the bindings between the API ports are specified.

For a given service composition, it is also possible to declare vertical bindings between API ports of the peer services and API ports of the service composition itself. This is useful whenever a service composition is published as peer service as well (see

next section). When publishing a service composition, the new API will be extracted automatically and put into the corresponding service advertisement.

6.5 Peer Service Execution Model

DEEVOLVE features three different execution models: single service execution, basic and distributed service composition. These will be explained now.

6.5.1 Single Service Execution

In this execution model, a consumer peer invokes a single peer service provided by a single provider peer. This peer service corresponds to a distributed application (FREEVOLVE) that has been augmented by an API and then published as a peer service.

During start-up, the DEEVOLVE peer environment is acting as the provider creates instances of the peer services it provides. It does so by parsing all available CAT-XML files. Based on the evaluation of the respective CAT files, the component structures of both the local composition and the interface parts are generated. A “DistComponent” object represents either composition parts. For the local composition part, the corresponding proxy structures are built as well. This proxy structure instantiates the defined components (accord to Java classes) and creates the bindings among them. The local composition part is now ready for use.

A peer consumer first has to locate the corresponding peer service advertisement. This advertisement contains all information needed to invoke the service. Before execution, the consumer has to subscribe for that peer service at the provider peer (see section 7.2.1 for more details). Both the execution and the subscription of a peer service can be initiated from within the DEEVOLVE console (section 6.6.3). Upon request, the provider returns a reference to the “DistComponent” object representing the interface composition to the consumer environment. Here, the corresponding proxy structures are created with respect to the component structures. This proxy structure of the interface composition then instantiates the complete components and establishes both the local bindings as well as the *remote bindings* between the façade ports. The proxies then register to the component structures at the provider site. This way, the proxies can be notified about adaptations made to the component structures (cf. section 8.4.4).

Each new instance of an interface composition is bound to an existing local composition. The consumers share a local composition and, thus, a peer service. This model is useful, if peers are willing to share the same state (e.g. for exchanging data). This is the default case, which is also compatible to the FREEVOLVE execution model. The execution model resembles the *shared object* execution model of the $SO_{P2P}A$ style.

If a peer service is marked as “not_shared”, each new request for a peer service yields to the instantiation of both a new local and a new interface composition part. This way, the consumer is associated with its own private peer service instance that is not shared with other peers. For both parts, the corresponding component and proxy structures are generated. The adaptation of the component structure only effects the adaptation to the bound proxy structure on consumer site. This execution model meets the *one service per customer* execution model of the $SO_{P2P}A$ architectural style.

For optimization purposes, a DEEVOLVE environment could also maintain a service pool, in which a predefined number of peer services instances are generated. This will limit the risk of too many instances being generated and, hence, limits the risk of re-

source exhaustion. This model corresponds to the *service pool* variant proposed in $SO_{P2P}A$. Although clearly a promising alternative, this model remains unimplemented in DEEVOLVE.

6.5.2 Basic Service Composition

This execution model presumes that a peer provider has defined a composite peer service of at least two different peer services. At least one peer service is provided by another third-party provider. In this model, bindings between the ports of the APIs of the participating peer services can only be established within the *consuming* DEEVOLVE environment (Figure 6-15). Due to this restriction, only bindings between the interface composition parts can be defined. The advantage of this model is that the provider peer does not have to provide resources for deploying third party peer services when a consumer is using the service composition. The consumer only is responsible for maintaining his own peer services.

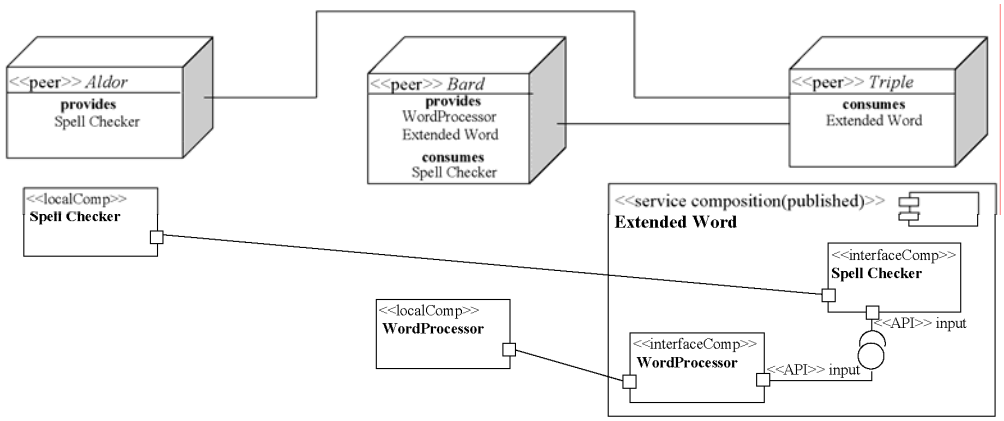


Figure 6-15: Execution model for basic service composition. A third-party peer ("Triple") consumes the service composition provided by peer "Bard".

In Figure 6-15, the operator of peer "Bard" has defined the service composition "Extended Word" out of the peer services "WordProcessor" (local) and "Spell Checker" (provided by peer "Aldor"). He then has published it as a peer service (see stereotype "<<service composition(published)>> in the service composition box. The operator of peer "Triple" has found the corresponding advertisement. From the implementation advertisement, he obtains the PeerCAT file for this composed service. By interpreting this PeerCAT file, the peer environment can establish bindings to peer "Aldor" and "Bard" in order to gain the interface composition parts of both peer services "Spell Checker" and "WordProcessor", respectively. According to the individual CAT files of both services, the internal components within a peer service are wired. Also, the remote bindings are set up. With respect to the given PeerCAT file, the "CompositionProxies" object is instantiated on peer "Triple" that now continues the instantiation of the composition. Basically, the proxy establishes the bindings between API ports of the interface composition parts within the environment of "Triple". These established bindings constitute the *added value* of both services: the word processor service can now use the spell checking service. Note the dependencies among the involved peers: peer "Triple" maintains connections to peers "Bard" and "Aldor". Peer "Bard" is not asked to maintain a connection to "Aldor". Recall that the "CompositionProxy" object registers to the object-oriented presentation of the PeerCAT structure (class "PeerServiceComposition"). This proxy resides at the consuming peer only.

Adapting the PeerCAT structure leads to the adaptation of the local proxy only and, thus, to the local service composition instance. All other instances are not affected. More information on this execution model can be found in the master thesis of Mitrov ([Mitrov, 2003]).

6.5.3 Distributed Service Composition

The distributed composition is an extension to the composition execution model explained in the previous section. In this execution model, no restriction is made concerning the binding of API ports. An API port of an interface composition part can also be connected with an API port of a local composition part *within a single environment* (Figure 6-16). The deployment of this execution model is then split across two peer environments. The benefit is that the requesting consumer peer is shielded from any unnecessary peer service interfaces. While the provider of the first execution model is completely ignored, he can be involved in the service execution.

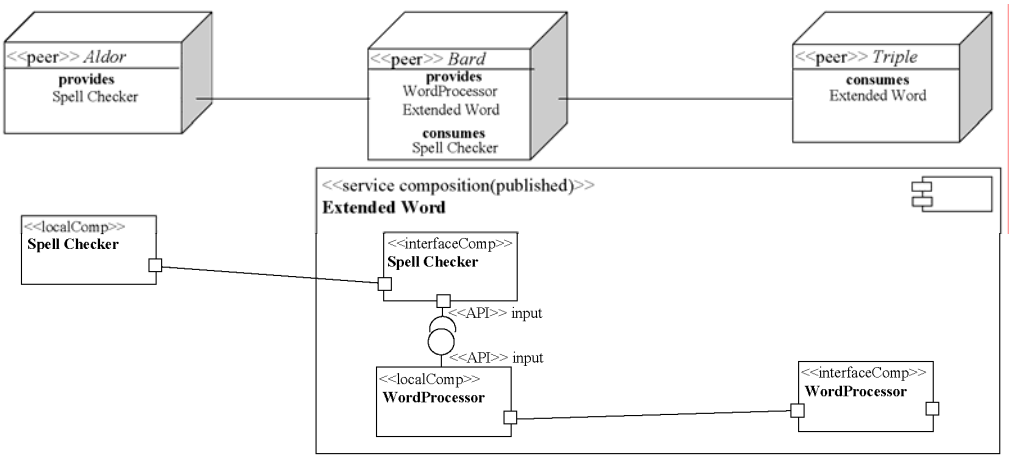


Figure 6-16: Execution model for a choreography composition model.

In this (perhaps more logical) composition model, consumer peer "Triple" only receives the interface composition part of peer service "WordProcessor". Peer service "Spell Checker" is bound with the local composition part of the "WordProcessor" service within the peer environment of "Bard". Thus, the interface composition part remains at the originating provider peer and is not migrated to the consumer peer.

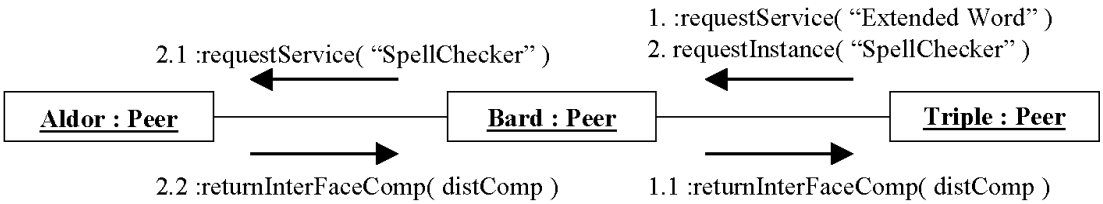


Figure 6-17: Communication between peers in the choreography execution model (UML communication diagram)

The construction of a distributed composition in Figure 6-16 starts again after the operator of peer "Triple" has retrieved the corresponding advertisement of the published service composition. According to the contained PeerCAT file, the DEEVOLVE environment of "Triple" requests the interface composition part of peer service "WordProcessor" (see Figure 6-17, step 1). Following that, the components are instantiated on both peers and the local and remote bindings are linked by the proxy objects.

The peer environment of “Triple” then detects a remote service binding between peer service “Spell Checker” and the local composition part of the “WordProcessor” service. This binding cannot be performed in the local environment of “Triple”. Therefore, the environment sends a request to peer “Bard” to bind in the “Spell Checker” service in the dedicated instance of the “WordProcessor” (step 2). To facilitate this binding process, the peer hosting the binding (here: peer “Bard”) creates a “CompositionProxy” object that takes over the process of deploying and binding all necessary services. This object remains at the provider site. For each request for a service composition, a pertaining “CompositionProxy” object is created. Thus, the provider can perform adaptations to *all* running instances of a given service composition. The consumer can access this proxy object remotely. He is able to adapt his single instance of that service composition. These changes are stored within the discovered PeerCAT file. The original PeerCAT file from the provider peer remains unchanged.

Since both service “WordProcessor” and service “Spell Checker” cannot be shared (type “not_shared”) each consumer gets its own instance of a local composition part. Therefore, the “Bard” peer also needs to request a new instance of peer service “Spell Checker” (step 2.1). The peer “Aldor” instantiates a new local composition part and returns an interface composition part referencing to that local part. If that peer service was shareable (type “shared”), an interface composition part would be returned that would point to the existing local composition part of peer service “Spell Checker”. Naturally, adapting a shared local interface may yield a dependency violation. A service provider should therefore analyze potential consumer dependencies (chapter 7). A consumer is not allowed to adapt a shared local part by default. An adequate access control model could clearly organize the access on shareable compositions.

6.6 Prototype Implementation of DEEVOLVE

In this section, the prototype of DEEVOLVE is presented. The prototype has been developed based on the Java 5.0 platform by Sun. Apart from the illustration of the layered architecture of a single local DEEVOLVE environment, some tools are presented.

6.6.1 Layered Architecture of a local Peer Environment

The prototypical implementation of a DEEVOLVE runtime environment adheres to an open layered architecture consisting of six different layers (Figure 6-18). The first two layers (seen from the bottom of Figure 6-18) are taken from the JXTA framework and represent the core functionality as well as the main services of JXTA (see [Sun, 2005b] for more information). From these layers, only those concepts have been adopted that are necessary for DEEVOLVE. Most importantly, the advertisement, the discovery, and the peer group mechanism reside on these layers. The implementation classes of the Java RMI protocol reside on the same layer as the core functionality. RMI is used by the services that reside on DEEVOLVE’s runtime layer. This possible because DEEVOLVE features an open architecture: any layer can invoke operations of services from *any layer below*. The DEEVOLVE core layer consists of important subsystems for monitoring dependent peers (used for exception detection, see 8.3.1), for subscribing to a peer service (section 7.2) and for sending and receiving messages between DEEVOLVE environments (section 6.6.4). Subsystems used to deploy and to manage compositions within a runtime environment are situated on the runtime layer. DEEVOLVE extends the tailoring mechanisms of FREEVOLVE by new operations ena-

bling users to tailor a service composition as well as single peer services at runtime. The most relevant subsystems for this work reside in the service layer. Here, subsystems are available for handling exceptions (section 8.4), for checking the integrity of a composition (section 8.2), and for analyzing consumer dependencies (section 7.3).

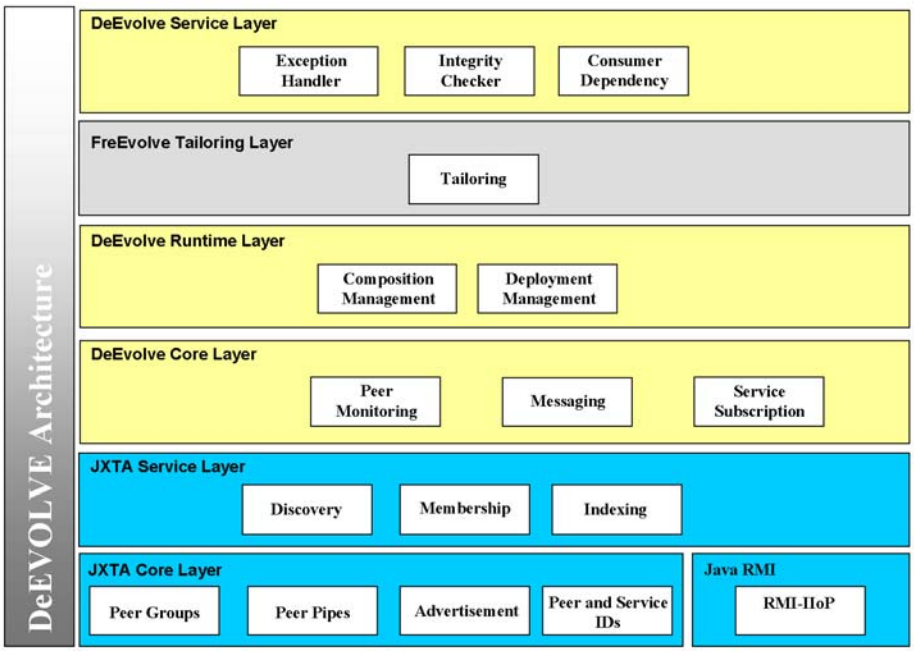


Figure 6-18: Visualization of the open layered architecture of a DEEVOLVE peer runtime environment. Any layer can invoke operations from any layer below

6.6.2 Starting the local DEEVOLVE Peer Environment

A DEEVOLVE peer environment can be booted from the command line console. The first dialog displayed is the configuration tool from the JXTA framework (see [Sun, 2005b], appendix B) for a snapshot and more illustrations). This tool allows for defining basic settings for the environment. Settings include, among other things, selection of peer type (rendezvous or simple peer), determination of other rendezvous peers (from a list of known rendezvous peers), the IP-address of the host, the peer name, or the indication of possible firewall servers. These settings only need to be entered once during the first initial boot process. After having finished the configuration process, the DEEVOLVE console appears, which will be explained in the next section.

6.6.3 The DEEVOLVE Console

The DEEVOLVE console is the central management application for a DEEVOLVE peer environment (see snapshot in Figure 6-19). By means of the DEEVOLVE console, an operator is capable of discovering new peer services (i.e. advertisements describing services). To do so, he is able to enter search keys in the upper-left text field. Asterisks can be used to extend the possible result set (e.g. “test*” delivers all possible matches starting with “test”). Search keys can refer to various properties of an advertisement, such as the service name, the creator name or a group name. Search queries are broadcasted to all available rendezvous peer, which– after potential query delegations to other rendezvous peers – send back a set of matching advertisements. The most rele-

vant attributes (e.g. name, version, provider, necessary groups) of these advertisements are summed up in the upper table.

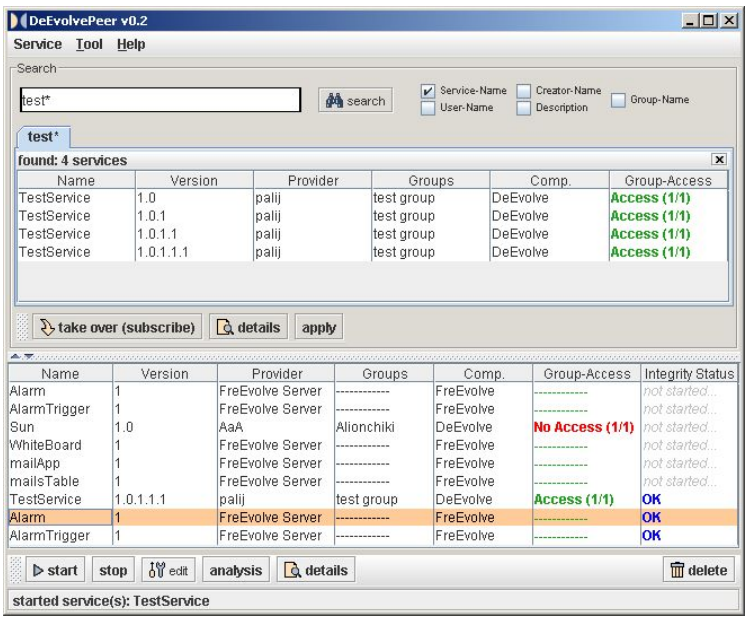


Figure 6-19: The DEEVOLVE console for managing the peer environment and its corresponding peer services

From the result set, an operator can take over or subscribe a selected peer service by pressing button “take over (subscribe)”. This operation drags the peer service in the lower table, where all subscribed and used services listed. It also sends a subscription message to the provider of that peer service indicating a new consumer dependency through the messaging service (see next section). The passed dependency level is, however, of lowest priority, it just signals an interest of the peer service. Later on, during usage, the operator can modify the dependency value (see section 7.2.4).

Having subscribed to a peer service, an operator is able to start a peer service (button “Start”). Each peer service is executed in its own *thread*. If the interface composition part of a peer service features dialogs or frames (components of Java Swing), then these are visualized and pushed to the front accordingly. At any time, the user is able to stop the service instance (button “stop”). In that case, the instance (i.e. the components of the interface part) will be removed completely from the memory and the remote bindings are deleted. The lower table of the console also contains the *local* service compositions. Like a single peer service, a service composition can be started by the “start button”. Depending on the arrangement of bindings, the services are deployed and executed (cf. execution models section 6.5). Row “integrity status” shows the current status of a service composition. If the status is “not_started”, then the service composition yet has not been started. If the status is set to “OK”, the service is running and every dependency on a remote peer is fulfilled. In the status “failed”, at least one dependent peer is unavailable and the composition is violated. More information about the functionality of the console can be found in [Palij, 2006] and [Mitrov, 2003].

6.6.4 The DEEVOLVE Messaging Service

All messages in DEEVOLVE are sent through the *DEEVOLVE messaging services*. This service allows any DEEVOLVE peer to send messages to any other DEEVOLVE peer.

Messages sent through this channel are simple XML-based documents. This service can dynamically bind various concrete communication layers (e.g. SMTP, Pipe Binding Protocol from JXTA). The SMTP protocol turned out to be somewhat efficient especially compared to the weak implementation of the PBP protocol (cf. section 2.2.5). An operator can use the DEEVOLVE Mail Client (Figure 6-20) for reading and sending messages. Besides, any tool of DEEVOLVE can use a simple API to subscribe as a listener for incoming messages that can be processed by the specific tool (e.g. the DEEVOLVE Analysis tool in section 7.4.1).

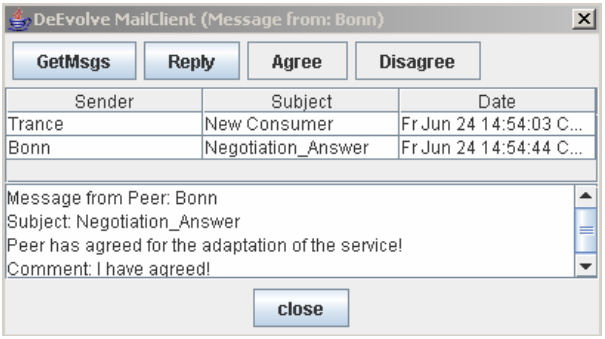


Figure 6-20: The DEEVOLVE Mail Client

The messaging service of DEEVOLVE serves as a concretization of the messaging service as formalized in the $SO_{P2P}A$ (see section 3.3.8.3).

6.6.5 Composition Tools

In [Mitrov, 2003], a textual composition tool has been developed that enables peer operators to declare service composition in a textual way (Figure 6-21).

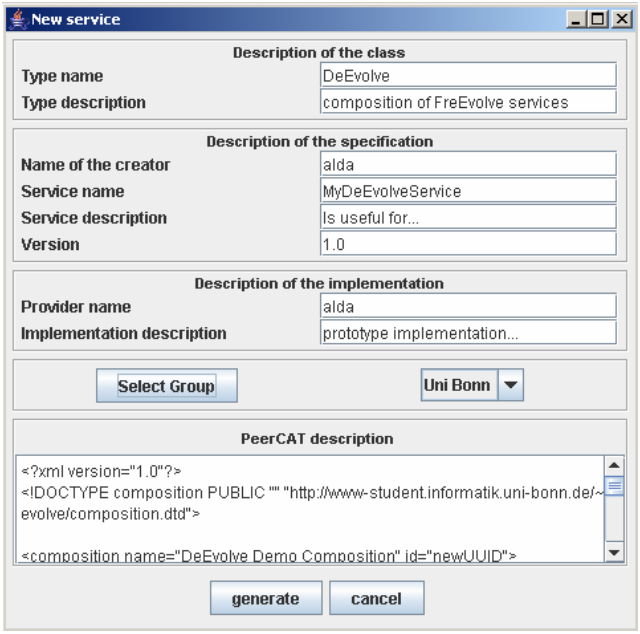


Figure 6-21: DEEVOLVE textual composition tool

If desired, the corresponding composition can directly be described by an advertisement and published within the peer-to-peer architecture. The new peer service then corresponds to a composite peer service. Besides defining the appropriate binding statement based on PeerCAT, a user is able to associate peer group memberships to a

composite service. If a consumer is willing to use this new peer service, he must be a member of all declared peer groups. This tool can also be used to re-load an existing composed service or local service composition in order to re-arrange (= tailor) the composition at runtime (see [Mitrov, 2003] for more information). For tailoring single peer services in a graphical, the TailorClient by Won can still be used (see [Krüger, 2002] for more details). A prototypical tool for creating service compositions in a graphical has been developed in the course of the research project but omitted here.

6.7 Evaluation

A first usability evaluation based on user observation has been carried out in [Mitrov, 2003]. In this work, the thinking aloud evaluation method proposed by Nielsen [Nielsen, 1993] has been applied to assess the usability of both the DEEVOLVE console and the textual composition tool. The evaluation of the prototype did also aim at estimating the user's comprehension of service-oriented peer-to-peer architecture. The overall reaction of the test person was satisfactory. All users understood the advantages of the distribution of peer services in decentral peer-to-peer network. The user rated the DEEVOLVE console as very useful. However, they complaint about the less intuitive textual editors and advised for a graphical tool. This advice has been taken into consideration through the development of the graphical composition tool.

In [Mitrov, 2003] the execution model for a basic service composition (section 6.5.2) has been developed. In the aftermath of this work, extensive discussions with local students and collogues as well as with researcher at conferences have revealed unclarity concerning this composition model. All persons claimed for a more straightforward model in which a service composition acting as a peer service only disposes a single front end (i.e. local composition part). All other front ends are to be hidden from the consuming user. These hints have eventually led to the addition of a third execution model – the distributed model – as described in section 6.5.3.

6.8 Conclusion

This chapter has presented the fundamental model of DEEVOLVE, the reference implementation for the proposed runtime environment of the $SO_{P2P}A$ architectural style. This chapter serves as the basis for the next chapters, in which the more specific parts of the platform covering notions for dependency management and exception handling are described. Related work comparing the platform with other approaches will be discussed within the next chapters.

Chapter 7
Consumer Dependency Management in DEEVOLVE

The $SO_{P2P}A$ architectural style proposes formalisms for end-user adaptation (tailoring) of peer services (section 4). The style divides this process of adapting a peer service into three disjoint stages. It starts with the analysis of potential consumer dependencies and – as a second step - proceeds with the ability to tailor the respective peer service specification. Eventually, the tailoring steps are delegated to both local and remote instances of the constituting composition parts of the modified peer service. The implementation of component-based adaptation methods makes use of the conceptual tailoring model proposed by Stiemerling's dissertation. Apart from some extensions to his model (cf. section 8.4.4), this work does not stress the adaptation model as its major contribution. This section mainly aims at describing the first part, namely the analysis of consumer dependencies in a service-oriented peer-to-peer architecture. Dependency analysis has not been covered by any previous work about the FREEVOLVE platform. Moreover, the proposed group-oriented approach for dependency management is also a novel approach compared to traditional approaches for dependency management for software architectures.

This section will highlight the concepts for substantiating the suggested formalisms for dependency management. These concepts have been turned into a prototype based on DEEVOLVE, the default implementation of the $SO_{P2P}A$ style. The last part of this section gives a brief overview of the realization of the component-based adaptation methods. The section ends with a survey on related work that compares other well-known approaches for dependency management with the proposed approach.

7.1 Overview on the Concept

The proposed analysis model that this approach is based on presumes two assumptions that are made in the $SO_{P2P}A$ style. The first underlying assumption implies that any consumer peer can subscribe to a list maintained by a service provider peer if the consumer peer relies on a public service offered by this service provider (section 3.3.10.6). Hence, a provider peer obtains an overview of all currently dependent consumer peers that use its public peer services. If the operator of a provider peer plans an adaptation, he is able to consult all subscribed peers before he carries out the adaptation. With respect to the $SO_{P2P}A$ style, an operator can augment a dependency on a used service with a value indicating the importance of that service in his setting (section 3.3.10.6). By intention, the definite range and semantics of those values remained

unspecified so that concrete peer-to-peer architectures are able to deploy their own, application-domain related values. This work proposes a couple of possible values from a catalogue that has been proposed elsewhere (see related work in 7.6).

The second assumption requires that peers have agreed to a common *adaptation policy* in the run-up to the adaptation of a service (see 3.3.9.1 and 4.1.2). The purpose of an adaptation policy is to clarify how a provider peer should handle existing consumer dependencies if he is willing to adapt a public peer service. Owing to the potentially huge number of peers involved in a peer-to-peer architecture, this approach appears rather unrealistic, at first glance. A provider peer would have to agree with all potential consumer peers that could possibly integrate its peer service. The underlying agreement process would certainly take an indefinite amount of time. For a more practical utilization, $SO_{P2P}A$ recommends that peers being regular members of a peer group agree upon a common adaptation strategy in the run-up to the adaptation of a service. A peer group thus prescribes an adaptation policy. New peers joining an existing peer group need to accept this adaptation policy. So, a peer willing to adapt a peer service only needs to announce the adaptation to consuming peers that are within a peer group. Again, the notion of an adaptation policy (and how to evaluate it) and that of an adaptation strategy remains rather unspecified in the proposed architectural style. This section will give some more meaning to those terms by proposing practical examples especially for adaptation strategies.

Before both fundamental concepts are elaborated in more detail in section 7.2 (consumer subscription) and section 7.3 (adaptation policy), their influence on the structural model of DEEVOLVE is outlined in the next section.

7.1.1 Extension to the Structural Model of DEEVOLVE

Figure 7-1 shows the refined structural model of DEEVOLVE as a UML class diagram to illustrate the concepts for dependency management explained so far.

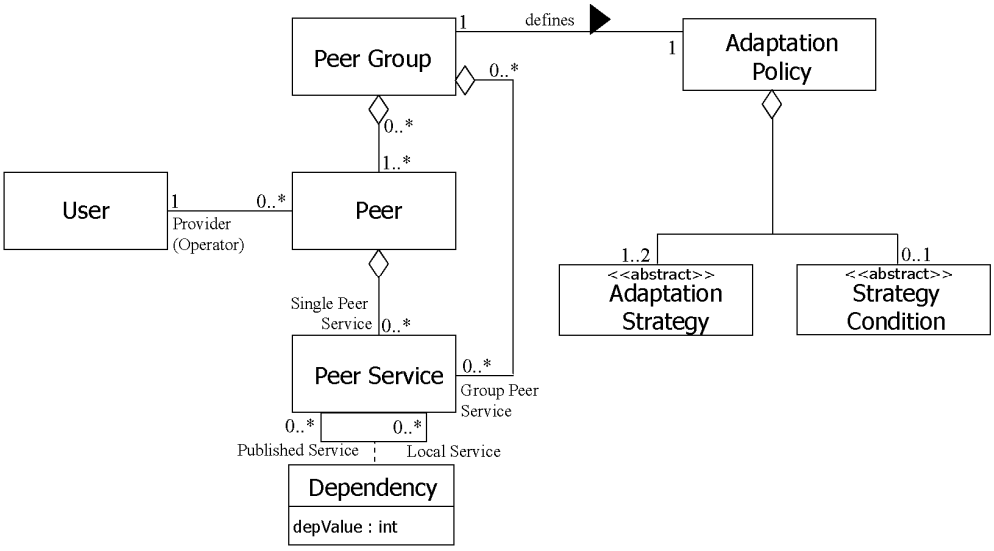


Figure 7-1: Refined structural model of DEEVOLVE including concepts of adaptation policy and subscription (UML class diagram)

The left part of Figure 7-1 visualizes the general structure of a service-oriented peer-to-peer architecture as it can be set up by DEEVOLVE (see section 6.2 for more explanations). The concept of a peer service has a new class association “Dependency”,

whereas the dependency relation is annotated by two roles “Local Service” and “Published Service”. A local service may hold dependencies to an arbitrary number of published peer services stemming from different provider peers. Local service means that the service is used internally and/or that it is published within the peer-to-peer architecture. This dependency defines no compositional binding element, but a *dependency value* that can be defined by consumers to indicate the importance or the relevance of a consumed peer (see section 7.2.3 for more details). The many-to-many association points out that a local service can have dependencies to many published services and vice versa. The association class “Dependency” is augmented by a dependency attribute *depValue* expressing the importance of a consumed peer service for a peer service.

A peer group is able to define an adaptation policy, which is denoted by class “AdaptationPolicy”. This class aggregates two further concepts, namely an adaptation strategy and an adaptation condition. These concepts are used to refine the imposed adaptation policy. Both are explained in more detail in section 7.3.

7.2 Subscription to a Peer Service

If a consumer is willing to use a public peer service, he has to subscribe for that service in the respective service provider’s environment. Unlike central architectures (e.g. like Web Services) subscription does not occur at a central directory (cf. UDDI directory). The subscription is done at the providers’ environment accomplishing them to maintain data on consumer dependencies internally. This data is used for controlling the adaptation of public peer services as explained later in section 7.3.3. At first, section 7.2.2 delineates the process of consumer subscription. Section 7.2.3 then explains the significance of the above-mentioned dependency values during subscription.

7.2.1 Process of Subscription

Figure 7-2 depicts a simple process model to explain the interaction between a provider and a consumer during the process of subscription. This model is a refinement of the more generic subscription model proposed in $SO_{P2P}A$ (section 3.3.10.6).

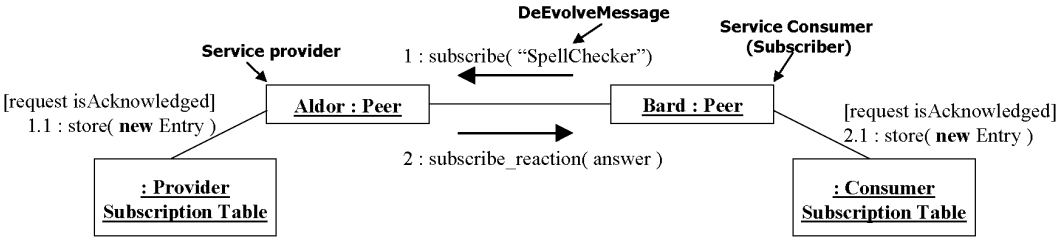


Figure 7-2: Process model for subscription (UML communication diagram)

If the operator of peer “Bard” is willing to subscribe to peer service “SpellChecker” provided by peer “Aldor”, he has to send a DEEVOLVE message to peer “Aldor” (step 1). The message contains the following parameters:

- *serviceID*: the unique identifier of the public service.
- *peerID*: the unique identifier of the public service (the UUID address)
- *peerName*: the name of the peer
- *depValue*: the dependency value indicating the relevance of the public service for the consumer (see section 7.2.3)

- *comment*: any comment that can be entered by the consumer prior to subscription
- *dependentServices*: indication, which local parts make use of the public service. This can either be a list of peer services or of compositions. This tag can be empty.
- *cascade*: states if the consuming peer wishes to be informed on occurred exceptions (yes) or not (no). This is later on used for exception cascading (section 8.3.5)
- *semantics*: this field can be filled with any kind of semantical information that could be useful for the provider. This data can be used later on evaluate integrity constraints (see section 8.2)

The initial subscription can be done by selecting a service from the table of found services within the DEEVOLVE console (see Figure 6-19). This user then has to press the button “takeover(Subscribe)” in order to drag the service to the lower table of selected services. During this step, a dialog is opened in which the consumer is able to rank the desired dependency value for that service and to enter a comment. After, a message is produced that contains all necessary data for setting up a subscription request (the rest is derived from the pertaining advertisement and from basic settings of the DEEVOLVE environment). Finally, the message is conveyed to the provider peer asynchronously. It arrives in the DEEVOLVE mail client in which the provider is able to read the subscription request (see example in Figure 6-20). The provider can use buttons “Agree” or “Disagree” to acknowledge or refuse the request, respectively. In both cases, a message is sent back to the consumer to announce the result of the request (step 2 in Figure 7-2). The message for sending back the result contains these parameters:

- *serviceID*: the unique identifier of the public service (useful for a consumer when he has to distinguish between many subscription requests)
- *acknowledged*: “yes” if the request has been accepted, “no” if not
- *comment*: any comment that can be entered by the provider prior to sending the result
- *cascade*: states if the providing peer also wishes to be informed on exceptions (yes) or not (no) that occur within the consumer environment. This is later on used for exception cascading (section 8.3.5)
- *semantics*: this field can be filled with any kind of semantical information that could be useful for the consumer. This data can be used later on to evaluate integrity constraints (see section 8.2)

Obviously, if the provider does not acknowledge the request, the fields “comment”, “cascade”, and “semantics” can be empty.

7.2.2 Storing Subscription Data

Subscription data is stored within the DEEVOLVE environment. As each peer environment may adopt the roles of consumer and provider at the same time, two different tables need to be maintained, that is, the *provider subscription table* and the *consumer subscription table,* respectively.

Provider Subscription Table

The provider subscription table is maintained by a service provider in order to store information on consumers. Principally, it stores all data on consumers that have subscribed for a distinct published peer service. For each consumer, a corresponding entry (*dependency object*) is created right after the request has been acknowledged (step 1.1

in Figure 7-2). A single dependency object of that list corresponds to a relation between one subscribed consumer and one local peer service. The object contains all data from the original subscription request message. A dependency object offers public methods for updating parameters especially used for updating dependency values. The table (in the prototype implemented as a class) also provides methods for obtaining aggregated views on data (e.g. to obtain all dependencies for a local service).

Consumer Subscription Table

The consumer table is maintained by a service consumer in order to save information on published peer services he has subscribed. It contains of the data that is passed back in the reply message of the provider expressing the result of the subscription request (step 2.1 of Figure 7-2). An entry is only created if the request has been acknowledged beforehand. The consumer subscription table allows service compositions to *override* internal dependencies to subscribed peer services (method `addDependency(Service,Dependency)`). This method is invoked after the creation of a composition, which is published as a *composite peer service* (see tools in section 6.6.5). In this phase, the final PeerCAT file is generated and put into the advertisement. In order to describe dependencies of a composition to externally used peer services, the necessary information can be entered in the selected tool. That information is stored in the dependencies tags in the corresponding PeerCAT structure (see Figure 7-3).

```
<composition name = "MyGroupware" ID = "anID" >
   <services> .... </services>
   <bindings> .... </bindings>
   <exceptionHandling> .... </exceptionHandling>
   <semantics> .... </semantics>
   <dependencies>
      <service id = "mySpellCheckerI" depValue = "Strong_Functional" dest = "MyGroupware"/>
      <service id = "myTool" depValue = "Low_Functional dest = "MyGroupware"/>
   </dependencies>
</composition>
```

Figure 7-3: Extract of a PeerCAT file for a Groupware composition. The dependency tags denotes internal dependencies between the local and used services

For each dependent service, the dependency value is stated (attribute *depValue*). After the table has received a request for updating a dependency value, it sends an update requests to the pertaining provider peers (see section 7.2.4). Placing the dependency value into the PeerCAT description and, thus, into the advertisement of the composite peer service improves the consumers' understanding of a service' structure. Knowing the structure is important, if a consumer tends to accept only simple services but no composite peer services.

Peer Store

The DEEVOLVE Peer Store is a class that integrates both the consumer subscription and the provider subscription table Figure 7-4. It provides a facade interface for both tables. This way, all insert and lookup operations are invoked from the peer store table, which in turn delegates the respective operation to the appropriate table. Furthermore, the peer store table implements methods for obtaining an aggregated view on both tables. Most notably, it offers method "`Peers[] getThirdPartyPeers(PeerService)`" for getting a set of consumer peers that are transitively dependent to a consumed peer service. This method is later on used during exception cascading (section 8.3.4) to identify consumer peers affected by the failure of a single peer service.

The “PeerStore” class also plays an important role during exception detection (see section 8.3). The concept of “PeerStore” will be elaborated during these sections.

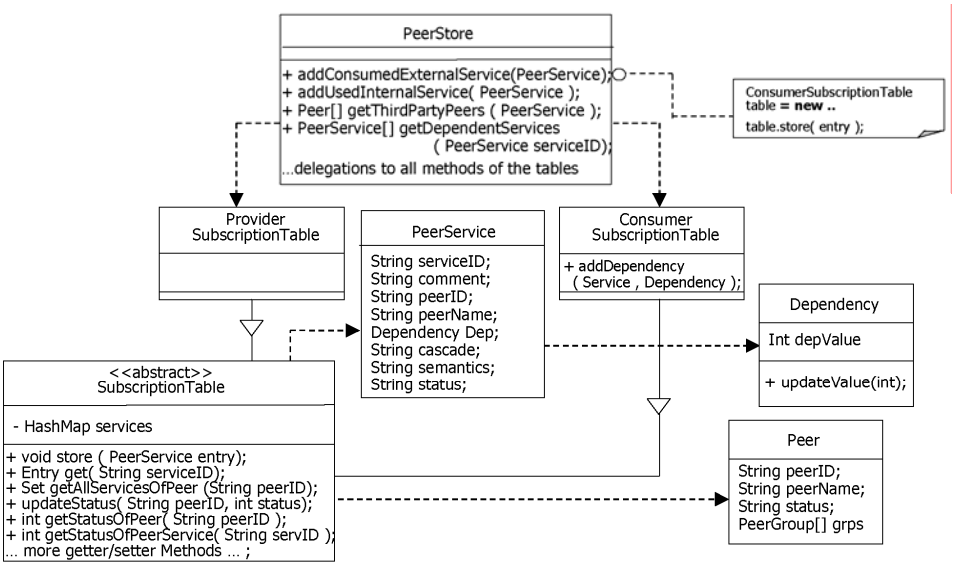


Figure 7-4: The DEEVOLVE PeerStore class for storing any kind of information on published and consumed peer services

7.2.3 Dependency Values

The consumer of a peer service is able to determine a rated value of each used peer service and to send this value to the provider of the consumed peer service. An initial value can be sent during the actual subscription procedure as explained above. A consumer can always update a value if necessary. With respect to the original architectural style $SO_{P2P}A$ there is no limit concerning the number of values. For a concrete architecture, however, it is good practice to have only a small number of values whereas each value has a concise semantic. The *five* dependency values presented in this section are based upon an extensive catalogue provided by Ensel [Ensel, 2001]. The purposes of these are explained in the following:

Not Determined Value

A peer operator (or consumer) declares just a dependency on a public peer service but yet no concrete importance value has been determined. Once again, to declare a dependency means that either a consumer uses a peer service directly or that he binds the service with other (local or remote) services towards a composition. This dependency value can be used for application settings, where no rated value is necessary. During the analysis of consumer dependencies, only the actual dependency is taken into consideration but no value.

Interest Dependency Value

The peer operator has indicated a public peer service as an *interest service*. An interest service is not directly used by local components but is designated for a later usage or for an upcoming composition with local or remote services towards a new application or new peer service. Hence, there is *no functional* dependency available. This dependency value is suitable if a consumer acting as component assembler first collects peer services before concrete compositions are defined. Most likely, he has to reason about

the given public interfaces of those services in order to see how they might be composed with other services. During this preparation period, it would be somehow beneficial, if these remained stable during that time.

Low Functional Dependency Value

The peer operator has composed a public peer service with other local or remote peer services towards a new application. That service is used directly by the peer operator himself. The peer operator has rated the importance of this functional dependency as *low*. This value can be applied in application scenarios with low demands on availability of the service (e.g. a grammar checker that is used once in a week). A failure or manipulation of that value would not cause serious damage. The justification of this dependency value is subjective, but enables peer operators to discriminate between a low and strong dependency (see next value).

Strong Functional Dependency Value

The peer operator has composed a consumable peer service with other local services towards a new application that is used directly by the operator himself. He also rated the importance of this dependency as *strong*. This value can be taken for application scenarios with high demands on availability and reliability on the consumed services (e.g. 24x7 hours / week). A failure or serious manipulation of that service would cause tremendous functional, project-related, or even financial damage.

Transitive Dependency Value

The peer operator has composed a consumable peer service with other local components towards a new peer service that has been, in turn, located and used by other third-party peers. Hence, there are transitive dependencies available within the topology of the peer-to-peer architecture.

The dependency values described here are sorted according to the *impact* the dependent peer service has within the local environment of the consumer. While the impact of the first dependency value (Not Determined) is low, the impact of the last value (Transitive Functional Dependency Value) is by no means high. Consequently, an arbitrary or unheralded adaptation of a public peer service with (at least) one transitive functional dependency would actually cause the most critical violations of behaviour within a peer-to-peer architecture. There is only one value for expressing transitive dependencies among peer services. This value could certainly be decomposed into further other values (e.g. strong transitive, interested transitive). The work omits this level of decomposition.

There is an implicit sixth dependency (No Dependency) stating that a consumer holds no or no longer a dependency on a peer service. This is actually an initial value that can be determined between a consumer and a public service. A consumer automatically updates from any of the five concrete values to this value whenever he unsubscribes from a public peer service. The next section 7.2.4 elaborates the processes of both updating a dependency value as well as unsubscribing from a service.

7.2.4 Updating a Dependency Value and Unsubscription

DEEVOLVE maintains each obtained dependency value describing the dependency between a local and a used peer service in an object of class “Dependency”. The update from one dependency value to another dependency value can be pictured in a semi-

formal way by means of a UML state diagram (see Figure 7-5). With respect to the idea of a state diagram, it is assumed that an object of class "Dependency" resides in exactly one state. The state is determined by the current dependency value expressed by a consumer. The dependency object can transition from that state to another one after the occurrence of an event at a consumer's environment. An event conforms to the push of a button to signal some behavior. Given some event, an action can be executed during the transition. This action is always executed at a provider environment. An action corresponds to a method call to invoke some behavior.

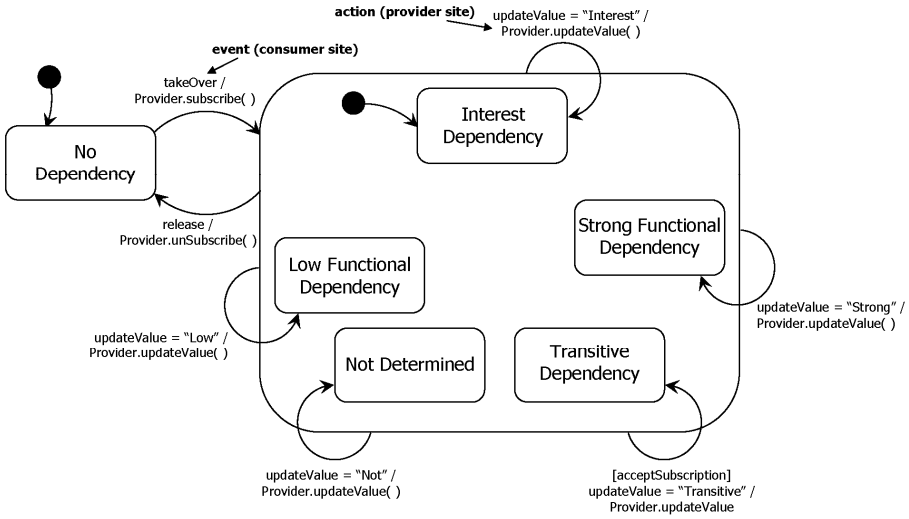


Figure 7-5: Semi-formal presentation of transitions between the states of a dependency object. A state is computed by a dependency value (UML state diagrams)

At start time, the dependency objects remains in the No Dependency state. If the consumer subscribes to a peer service (button "TakeOver" from the DEEVOLVE console), then the state switches to state Interest Dependency that is part of a composite state[20]. During that state transition, the subscription routine is invoked at provider site (see also section 7.2.1). At any time, the consumer is capable of updating the value of a subscribed peer service (button "UpdateValue" from the DEEVOLVE Console) to a new value. This event invokes the update routine within the environment of the peer consumer. Likewise to the event of a subscription request, the provider obtains a DEEVOLVE message by the consumer including the updated value. If the provider accepts it, the consumer table is updated accordingly (action *Provider.updateValue*).

Again, a consumer is able to take the public service and to integrate it into a local composition and publish this composition as a composite peer service. From now on, the consumer acts as a provider of service as well. A transitive dependency is given, if some other third-party peer has located that composed peer service and makes use of it. Consequently, this third-party consumer sends a request for service subscription to the provider. If the provider of the new service accepts the subscription ("Accept" button of the DEEVOLVE mail console), the provider is asked to update the dependency value to Transitive Dependency. After having acknowledged the subscription (modeled as condition *acceptSubscription*) and after the event of updating the value, an update message is sent to the first provider. The provider then updates the value too.

[20] The reader should refer to the UML 2.0 user guide ([Booch *et al.*, 2005], chapter 25) to obtain an overview of the semantics of composite states and of state diagrams in general.

At any time, the consumer is facilitated to unsubscribe from a peer service (button “Release” in the DEEVOLVE console). Right after, a message for unsubscription from the service is conveyed to the provider (see also previous section). This signals the provider environment to invoke behavior for unsubscribing the user and to set the value back to No Dependency. The consumer remains in the list at first. The provider is capable of deleting the object after a while manually.

Up to this point, operators of provider peers are yet facilitated to obtain an overview of dependent peers consuming their provided peer services, but there are neither rules that indicate how to handle dependencies nor rules that dictate when an adaptation can be proceeded with or not. As mentioned earlier, this work proposes the adoption of an *adaptation policy* for realizing such rules for provider peers. Adaptation policies also take into account the different dependency values between consumer and provider peers in order to differentiate between less important and highly crucial dependencies. The following section envisions the concept of adaptation policy.

7.3 Adaptation Policy

The right part of the structural model of Figure 7-1 exemplifies the concept of adaptation policy as one of the integral parts of the DEEVOLVE platform. Generally speaking, an adaptation policy provides rules for each service-providing peer concerning how to handle service dependencies before adapting a public peer service. As one can see in Figure 7-1, each peer group is associated with one adaptation policy. It is assumed that the originators or the maintainer of a peer group have specified the conditions of the policy in advance. New peers willing to join a group are forced to accept these conditions as specified by that policy. With respect to the architectural style $SO_{P2P}A$, policies are by no means fixed but can evolve during the lifetime of peer groups.

An adaptation policy is represented as an aggregation of two concepts, that is, an *adaptation condition* and an *adaptation strategy*. An adaptation strategy denotes an explicit procedure describing *how* a service-providing peer has to proceed in case of dependent consumer peers. An adaptation condition dictates *when* a selected strategy can be executed. Both concepts will be treated separately within the next two sections.

7.3.1 Adaptation Strategies

DEEVOLVE features four different adaptation strategies. All conform to specialized concepts of the abstract concept “AdaptationStrategy” of the structural model in Figure 7-1. These strategies are appropriate for applications scenarios, in which peer services are used to support the collaboration of dispersed working users. Owing to their *omnipresence*, users can be actively involved in the effectuation of the adaptation strategies. Each strategy is elaborated in the following.

Conservative Adaptation

An adaptation can only be executed if *no* dependencies are available. This strategy presumes no consultation with dependent consumer peers. It can be applied to application structures with high demands on availability (24x7) and reliability or with only little maintenance support.

Negotiation before Adaptation

An adaptation can only be carried out if all dependent consumer peers have been *consulted*. Moreover, all consulted peers have to *acknowledge* the adaptation request. This strategy can be utilized in application scenarios in which many high-value dependencies on consumer peers can be expected.

Notification before Adaptation

An adaptation can only be carried out if all dependent consumer peers have been *notified* in advance. Note that there is no approval necessary by the consumers. This strategy can be applied in scenarios with high maintenance support or with a high degree on user presence. A circumventive reaction to adaptation requests should be expected.

Liberal Adaptation

An adaptation can be carried out directly without any notification of or consultation with dependent peers. This strategy can be applied to scenarios with only little or even no degree on decentralization.

Adaptation strategies preceding negotiation or notification assume communication between the service provider and all pertaining service consumers. While notification necessitates a uni-directional channel, negotiation requires the establishment of a bi-directional channel between provider and consumers. A bi-directional channel can be used many times in either direction, so that provider and consumers can debate, for instance, on exact details or the scope of the planned adaptation. The final response comes from the consumer by sending a consent or a denial to the original request. Both notification and negotiation are realized by the DEEVOLVE mail tool.

As a base for the formal representation in section 7.3.3, the above-mentioned adaptation strategies are treated as members of set *AdaptStrat:*

$$AdaptStrat = \{Conservative, Negotiation, Notification, Liberal\}$$

This suggested *set* of adaptation strategies is not fixed but can be extended (see more information in section 7.4.4). The DEEVOLVE prototype assumes a *priority relation* among the members of that set. This relation postulates that liberal adaptation has the lowest priority while conservative adaptation obtains the highest one:

$$Liberal < Notification < Negotiation < Conservative$$

Prioritizing strategies is later on useful for performing the analysis of dependencies, especially when a peer service belongs to more than two peer groups with different adaptation policies (section 7.4.3). In that case, the policy with the higher priority is chosen. The priority relation is a global assertion. If a new adaptation strategy is formulated within a peer-to-peer architecture, all affected stakeholders of it must come to a mutual agreement concerning the priority of the new strategy. That new assertion must then be made available to all peers. This process, however, is not further regarded (and implemented) in this work.

7.3.2 Adaptation Condition

The adaptation strategies as described above are quite restrictive at first sight. Given a selected strategy (e.g. Notification), that particular strategy must always be applied independent of the actually available dependencies. An adaptation condition aims at

weakening that rule by introducing a *pre-condition.* This condition dictates when an adaptation strategy is to be executed or when a liberal, that is, immediate adaptation can be assumed instead. An adaptation condition always introduces a *stronger* pre-condition. A stronger condition allows for *fewer cases* in which an adaptation strategy can be applied. At the same time, it allows for *more cases* in which the affected peer services can be adapted immediately.

The rule for evaluating an adaptation condition states that the main strategy is applied if the condition is *false*. Liberal adaptation is assumed if the condition is *true*. Instead of liberal adaptation strategy, a designer of an adaptation policy can use another alternative adaptation strategy with *a lower priority*. An adaptation condition, thus, enables designers of an adaptation policy not only to impose one *main* adaptation strategy but also to involve an *alternative* strategy (see section 7.3.3). If no alternative adaptation is provided in the definition of an adaptation policy (see section 7.3.4), then DeEvolve supposes the liberal adaptation strategy as the default alternative.

The specification of an adaptation condition can be formulated in terms of both the number of current dependencies on a public service and the current dependency values available for these dependencies. A target variable *tv* is to be declared in order to state when a condition is fulfilled or not. A target usually represents an integer or real number (e.g. expressing the number of current dependencies). The operation to relate the target value with the actual values is indicated separately. Although the number of conditions is not restricted, just a small set of operations is assumed:

$$operation \in Operation = \{greater, less, equals, not\ equals\}$$

In analogy to the $SO_{P2P}A$ style (section 4.1.2), an adaptation condition can also be rephrased as a function that takes all registered remote peer services *rs* that are dependent on a consumable service *ls* provided by a peer with URN *i*[21]. All dependent services and the local service belong to peer group with URN *m* (*PS* denotes the set of all peer services):

$$adaptCond_m : \quad PS \times PS^n \times \Re \times Operation \rightarrow Boolean,$$

$$adaptCond_m(ls^m_{i,j}, rs^m_{i,j,1}, ..., rs^m_{i,j,n}, tv_m, operation_m) = \begin{cases} true, & if\ condition\ is\ fulfilled \\ false, & else \end{cases}$$

Function $adaptCond_m$ denotes the adaptation condition that was previously defined and prescribed by a peer group with unique ID *m*. The term $rs^m_{i,j,k}$ denotes the *k-th* of *n* remote services that are dependent on the *j-th* public peer service of peer with URN *i*. The term *condition* on the right side of the function represents the internal rule of the adaptation condition. If this condition is fulfilled, then the function returns true. If not, then the function returns false.

Auxiliary Functions

The function $adaptCond_m$ itself is able to apply various auxiliary functions to see if the condition is fulfilled. For instance, the function *value*($rs^m_{i,j,k}$) returns the corresponding (numerical) dependency value of the depending remote service $rs^m_{i,j,k}$. Function *deps*($ls^m_{i,j}$) returns the total number of dependent remote services for a local service

[21] Apparently, the function-oriented formulation of adaptation conditions has no surplus value compared with the formal presentation of the adaptation policy concept in $SO_{P2P}A$. This works, however, claims that it is more practical to use functions in order to *better* clarify the two concrete adaptation conditions later on in the section.

$ls^m_{i,j}$. These auxiliary functions constitute the fundament for describing the internal condition of function $adaptCond_m$. Two examples provided in the next paragraphs. The concrete examples conform to specializations to the abstract concept (class) "AbstractCondition" as pictured in Figure 7-1.

Example of an Adaptation Condition 1 (Weighted Average)

The following example of an adaptation condition computes a weighted average of all dependency values available for subscribed peer services. If the result is lower than *tv* (here: 5), the condition is true:

$$adaptCond_m(ls^m_{i,j},..,rs^m_{i,j,k},..,5,"<") = \begin{cases} true, & if\ \frac{1}{deps(ls^m_{i,j})}\sum_{k=1}^{n} value(rs^m_{i,j,k}) < 5 \\ false, & else \end{cases}$$

Example of an adaptation condition 2 (Maximal Number of Dependencies)

The next example of an adaptation condition prescribes a maximum number of dependencies to a public service.

$$adaptCond_m(ls^m_{i,j},..,rs^m_{i,j,k},..,"16","<") = \begin{cases} true, & if\ deps(ls^m_{i,j}) < 16 \\ false, & else \end{cases}$$

Hence, if a public peer service holds less than 16 dependencies to other consumer peers, then the condition is true and, consequently, the liberal adaptation strategy allowing the immediate adaptation of a peer service is applied. This assumes, of course, that no other alternative adaptation strategy has been selected. A more restrictive version of this condition is $deps(ls^m_{i,j}) = 0$. This condition entails that an adaptation of a peer service can only be started when *no* dependency from a remote service is at hand. This special adaptation condition is *implicitly* applied to the "conservative" adaptation strategy. This implicit condition is hard-wired within that strategy, that is, it is not possible to override it by an external adaptation condition.

7.3.3 Evaluation of an Adaptation Policy

The process of evaluating an adaptation policy comprises the steps of checking whether an adaptation is true or false and of determining the appropriate adaptation strategy. For instance, if *adaptCond* function in example (2) delivers *false* as a result, then the adaptation strategy selected by a peer group can be put into effect. This strategy is termed as the *main* adaptation strategy. If, however, *adaptCond* delivers true, then an *alternative* adaptation strategy can be assumed. The default adaptation strategy is the "liberal" adaptation strategy if not otherwise defined. The computation of the adaptation policy can be phrased as a function *compPolicy*:

$$compPolicy_m: \quad Boolean \times AdapStrat \times AdapStrat \rightarrow AdapStrat,$$

$$compPolicy_m(\ adaptCond_m, strat^{alternate}_m, strat^{main}_m) = \begin{cases} strat^{main}_m, & if\ adaptCond_m(...) = false \\ strat^{alternate}_m, & else \end{cases}$$

The alternative strategy must have a lower priority as the main strategy. A refinement of function *compPolicy* would be, to assume the adaptation strategy with the next lower priority for the case when function *adaptCond* yields false. This option is presented in [Alda, 2005a] but not further detailed here.

For a given peer service, the adaptation policies of all peer groups that are specified for *this* peer service have to be analyzed. Consequently, an adaptation of a peer service can only be performed if all strategies have eventually been obeyed (e.g. all dependent consumer peers have been notified). Assume a function *recheckPolicy* that determines, whether a peer operator has adhered to a strategy:

$$recheckPolicy_m : \quad AdaptStrat \rightarrow Boolean,$$

$$recheckPolicy_m(\ strat_m) = \begin{cases} true, & if\ strat_m = "Notification" \wedge user\ notified\ all\ consumer\ peers \\ true, & if\ strat_m = "Negotiation" \wedge user\ consulted\ all\ consumer\ peers \\ & \qquad \wedge all\ request\ were\ acknowledged \\ true, & if\ strat_m = "Conservative" \wedge no\ dependencies\ are\ available \\ true, & if\ strat_m = "Liberal" \\ false, & else \end{cases}$$

The rule expressing when an adaptation on a local service can be carried out is to be phrased as follows (let the set $Strategies_i$ include all strategies that have been identified for the local service with index *i*):

$$Localservice\ ls_{i,j}^{m}\ can\ be\ adapted :\Leftrightarrow$$

$$\forall\ (strat_m) \in Strategies_i : recheckPolicy(strat_m) = true$$

For a peer service that associates many peer groups with adaptation policies and, thus, strategies of different order, the strategy with the highest order has preference.

A more technical presentation on how exactly the analysis is organized and performed in the DEEVOLVE platform is outlined in section 7.4.2.

7.3.4 Advertisement of an Adaptation Policy

An adaptation policy is declared and published as a separate section within a peer group advertisement (see section 6.2.4). The structure is illustrated in Figure 7-6.

```
<jxta:PGA>
   <GID> urn:jxta:jxta-DeEvolveGroup</GID>
   <Name> DeEvolveGroup </Name>
   ...
   <AdaptationPolicy>
      <Strategy>
         <type order = "main" >Notification</type>
         <class> org.deevolve.analysis.strategy.Notification </class>
      </Strategy >
      <Strategy>
         <type order = "alternate" >Liberal</type>
         <class> org.deevolve.analysis.strategy.Liberal </class>
      </Strategy >
      <Condition>
         <type>WeightedAverage</type>
         <class> org.deevolve.analysis.metric.WeightedAverage </class>
         <operator> lower </operator>
         <value> 5 </value>
       </Condition>
   <AdaptationPolicy>
</jxta:PGA>
```

Figure 7-6: The structure of an adaptation policy within a group advertisement

Within the tag AdaptationPolicy, the adaptation strategies and the belonging condition are specified. In the example of Figure 7-6, the strategy “Notification” is selected as the main strategy. “Liberal” adaptation is chosen for the alternate strategy. The indica-

tion of the alternative strategy could be omitted since the "liberal" adaptation strategy constitutes the default adaptation strategy. In the class tags, the class representing the strategies during dependency analysis is indicated (see section 7.4.3 and 7.4.4 for more details). The condition for determining the appropriate strategy is "WeightedAverage". This condition coincides to the formal specification of the weighted average as brought in section 7.3.2. Again, this condition dictates, under which condition the *main* strategy is to be chosen. Tag operator defines the actual operator for the given condition (here: lower) Tag value holds the target value of the adaptation condition ("5"). If the condition is true, that is, the weighted average of dependency values to a peer service is lower than 5, the user is able to tailor the peer service. In this case, the alternative strategy is applied ("Liberal Adaptation"). If the condition is not satisfied (i.e. the weighted value is greater than or equals 5), the main adaptation strategy is applied (i.e. "Notification before Adaptation"). Then, the user must first notify all dependent remote peers before he is able to proceed with tailoring of that peer service.

Any peer that locates a peer group advertisement is able to extract the information concerning an adaptation policy. If the operator of a peer agrees to the demands of a policy (and of course to any further implications made by a peer group), he is asked to apply and finally to join to that group (see section 6.2.3). After receiving a confirmation of the join request from one of the group founders, the pertaining adaptation policy is put on a list containing all adaptation policies valid for a given peer environment. When a dependency analysis is carried out, exactly these policies are taken in order to compute the result.

7.4 Design and Prototypical Implementation

This section outlines more information on both the design and the prototype implementation of the presented approach for analyzing consumer dependencies in DEEVOLVE. At first, a part of the prototype for visualizing consumer dependencies is demonstrated (section 7.4.1). Afterwards, section 7.4.2 outlines the process of analyzing consumer dependencies in DEEVOLVE.

7.4.1 Visualizing Consumer Dependencies

The operator of a peer environment is always able to visualize the currently available consumer dependencies for each of his public peer services. For doing so, he marks a public peer service from the list of available peer services (lower table from the DEEVOLVE console, see Figure 6-19). Then, by pressing button "Analyze", the DEEVOLVE Analysis tool is opened that visualizes all dependencies for the selected local peer service (Figure 7-7). The visualization of consumer dependencies in the tool of Figure 7-7 is graph-based. This graph consists of three types of nodes:

- a node type for visualizing the local (published) peer service (blue filled circle)
- a node type for depicting the dependent remote services (this could clearly be a local composition also; there is actually no distinction made) deployed by third-party consumer peers (blue filled squares)
- a node type for visualizing peer groups (yellow filled circles)

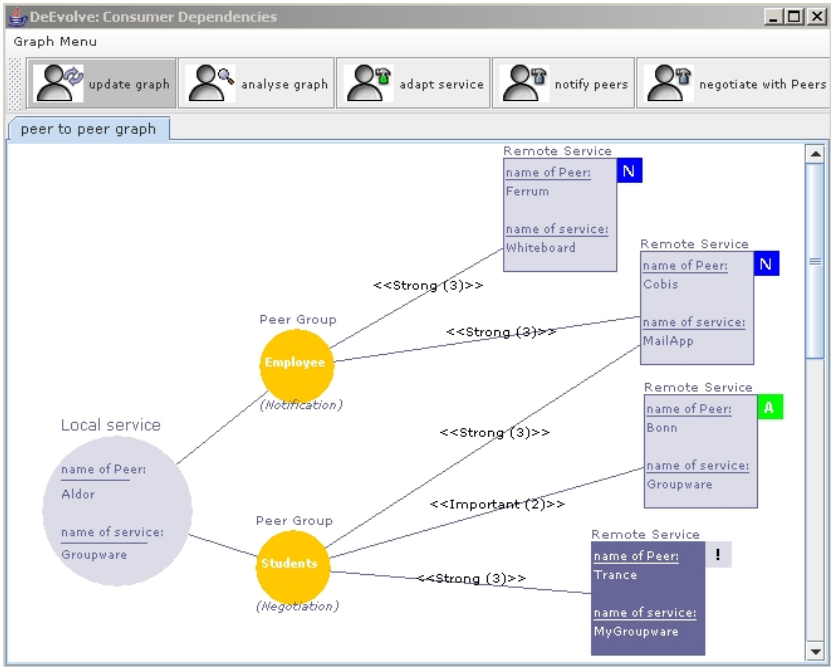


Figure 7-7: DEEVOLVE Analysis tool for visualizing consumer dependencies

A group node is also annotated with the main adaptation strategy the dedicated peer group is imposing. Each service node is augmented by the name of the hosting peer. The edges between peer group nodes and remote service nodes indicate a dependency relation from the remote peers to the local service. The insertion of peer group nodes aims at simplifying the readability of the graph. An edge is annotated with the corresponding dependency value declaring the relevance of the dependency (here, <<important>> denotes a low value). The third-party consumer peers have passed this value. The DEEVOLVE analysis tool is registered as a listener to mail events in the DEEVOLVE Mail service (see section 6.6.4). This allows the analysis tool to update the dependency values promptly when any consumer peer has sent a new update message. Alternatively, the user is able to update the graph by pressing the “UpdateGraph” button. This way, the analysis tool queries for new messages in the DEEVOLVE messaging service.

In the example of Figure 7-7, a local service “Groupware” is used by four different consumer peers. These peers have integrated this service as part of their own local peer services or compositions (“Whiteboard”, “MailApp”, “Groupware”, and “MyGroupware”). Each consumer peer either belongs to peer group “Employee” or “Students” except for peer “Cobis” (hosting service “MailApp”) that belongs to both groups. Each consuming peer has also indicated a dependency value.

The graph is produced mainly by means of the subscription data that has been conveyed by the consumers before. The analysis tool therefore makes use of the public interface of the PeerStore table of the DEEVOLVE environment (section 7.2.2). For each first-class concept (e.g. peer group, local service, and remote service) adequate classes are available. The corresponding class diagram accords to the principle structural model of Figure 7-1. After having obtained the data from the PeerStore table, an object-oriented model is produced featuring objects of these classes. From a separate list, objects representing the concepts of an adaptation strategy and adaptation condition are generated and added to the object tree (see more information in section 7.4.4).

The analysis of the graph can be initiated by pressing button “AnalyseGraph”. The analysis of the graph is presented in the next section.

7.4.2 Analyzing Consumer Dependencies

Based on the given visualization of the dependency graph, the peer operator can run an analysis of the dependencies to determine if an adaptation can be executed. The result of the analysis is presented in terms of a textual wizard (see Figure 7-8). The benefit of such wizard is that even unskilled users are able to comprehend the result of the analysis. The wizard also contains of a systematic procedure containing steps how to proceed towards a possible adaptation of the selected local peer service.

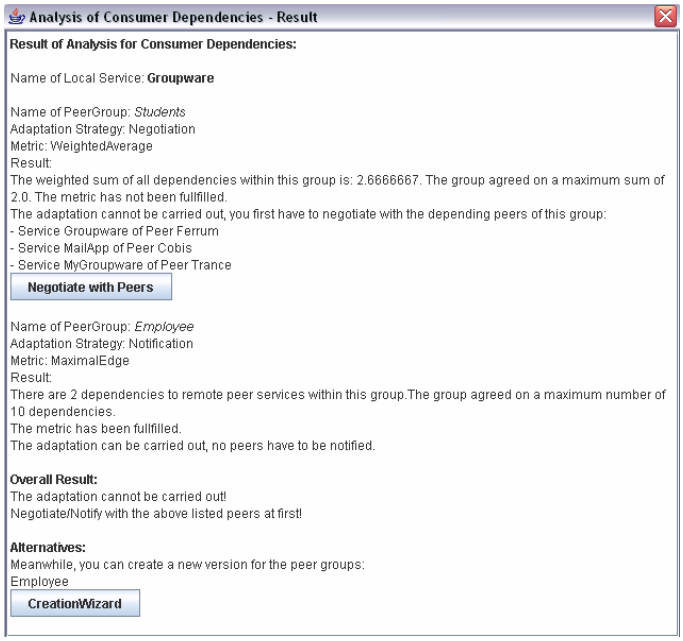


Figure 7-8: The result of the dependency analysis in form of a textual wizard

The result of the analysis is an aggregation of many partial results (see sequence diagram in Figure 7-9). First, intermediate results are obtained from the evaluation of the adaptation policies from each involved peer group. To do so, the analysis tool executes the analysis of the adaptation condition encapsulated in each "PeerGroup" object by iteratively invoking the method *analyzePeerGroup()*. This method calls the *executeComputation*() method from class "AdaptationCondition" to receive the result of the analysis. In this method, object "AdaptationCondition" is able to make use of methods *getNumOfDeps*() and *getDepValue*(RemoteService service) from the "PeerGroup" reference for obtaining information on dependencies values from consuming services. These methods serve as the implementation of the above-explained auxiliary functions *deps* and *value* (see section 7.3.2). Given a result of a condition, the "AdaptationStrategy" object can interpret the result. The interpretation is returned as a string (result). Both the result of the computation and the interpretation (i.e. the result of adaptation policy) are sent back to the "AnalysisObject" as string "finalResult".

After having obtained all results from the peer groups, the overall result can be computed. In principle, all final results of the given peer groups are concatenated. Based on the concatenation, the overall result can be derived. If at least one evaluated adaptation policy prescribes an adaptation strategy with a higher priority higher than "Liberal Adaptation", then the adaptation cannot be carried out. In this case, the textual analysis may propose alternative procedures in order to bridge the time between the analysis and the actual adaptation (e.g. creating a new temporary version while

leaving the old version as is). These alternative procedures can be obtained by the corresponding "AdaptationStrategy" objects (method call *getAlternatives*()).

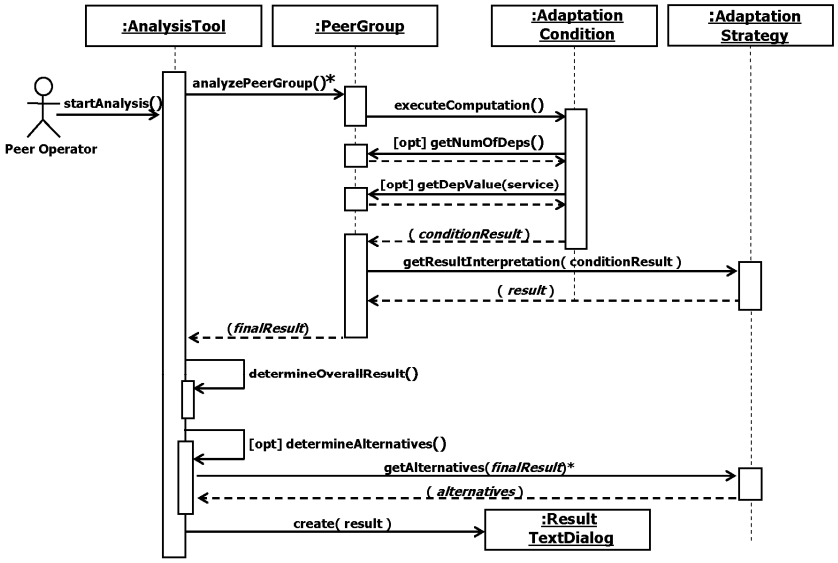


Figure 7-9: Process of performing the analysis of consumer dependencies (UML sequence diagram)

In the example depicted in Figure 7-8, the policy of peer group "Employee" allows for the immediate adaptation, while the policy of peer group "Students" requires the negotiation with all dependent peers. Thus, the overall result points out that the adaptation cannot be carried out, as there is at least one policy that does not allow an immediate, that is, liberal adaptation. Each result of a peer group indicating "Notification" or "Negotiation" follows a wizard part that entails further steps to execute the adaptation strategy. Such a wizard is decorated with a button, which facilitates a peer operator to execute the individual steps promptly (e.g. to notify the peers). In the example, the result offers the negotiation with dependent peers of peer group "Students". From the adaptation strategy of group "Employee", an alternative, optional procedure is proposed stating that a new version of the local service just for that group could be generated. It is up to the operator to decide whether this option is put into effect.

7.4.3 Executing the proposed Adaptation Strategy

After the analysis process, the operator is asked to carry out the proposed adaptation strategies. While both strategy "Liberal Adaptation" and "Conservative Adaptation" are silent strategies, the strategies "Notification" and "Negotiation" require the direct reaction coming from the user. Both strategies can be executed either from within the analysis wizard or through the visualization screen (see Figure 7-7, button "Notify peers" and "Negotiate with peers"). In either ways, the operator is able to enter the rationale for the adaptation in a textual dialog (e.g. the reason for the adaptation, the scope, details of the adaptation, affected API ports etc.). This rationale is sent to the consumers. During negotiation, both consumers and the provider are able to discuss upon this rationale. For this discussion, the mail tool of DEEVOLVE can be used.

The current status of the adaptation strategy can be obtained in two ways. Firstly, all notified or consulted peers are highlighted in the graph-based visualization (upper-right little square of the remote service node). "N" means that the corresponding peer has been notified (used for the notification strategy). "A" means that the consumer

peer has been consulted and that its operator has acknowledged the adaptation request[22]. "!" means that the consumer peer has been consulted but no acknowledges has yet been received. The two latter symbols are used for the negotiation strategy. Secondly, the user is able to start the analysis process again. Then, the current status of the already started adaptation analysis is taken into consideration during the analysis. If the user has followed all strategies, the adaptation can be started. In the example of Figure 7-7, the adaptation cannot be started. All peers of group "Employee" (imposing notification before adaptation) have been notified. From the peer group "Students" (imposing negotiation before adaptation) only one peer ("Bonn") has sent an acknowledge. The acknowledge for peer "Trance" is still pending. Peer "Cobis" still needs to be consulted although it has already been notified. Since this peer is a member of two groups with different policies, the policy with the higher order ("Negotiation") has precedence. Again, any update of the graph is recognized and visualized directly.

7.4.4 Extensibility Mechanisms

DEEVOLVE allows the addition of new adaptation strategies and conditions to the already given set of strategies and conditions explained in this work. A new strategy must extend the interface "AdaptationStrategy" that declares the relevant methods a concrete adaptation strategy class must implement[23]. A condition must extend interface "AdaptationCondition". For both interface, predefined abstract classes are available. The most relevant method of "AdaptationCondition" is *executeComputation*() that carries out the analysis of the condition and returns a result (object of class "ConditionResult"). For "AdaptationCondition", method *getResultInterpretation*() is important to obtain the interpretation of the result from the perspective of the strategy. Both methods need to be implemented by a developer. Once again, the process concerning how to introduce a new condition or strategy within a community is not addressed.

7.5 Tailoring a local Peer Service

After the analysis has been completed and adaptation strategies have been followed, the actual adaptation can be started (button "adapt Service"). This button is disabled as long as the analysis is not completed. By pressing that button, the "TailorClient" is opened. This tool enables an assembler to tailor the peer service in a graphical way. The functionality of the "TailorClient" is omitted here (see [Won, 2004] for details).

When the tailoring activity is finished, DEEVOLVE produces a new advertisement for the adapted peer service and publishes it through the architecture. The version of the service is increased automatically. Any consumer peer that retrieves such advertisement is able to bind this peer service. Registered (dependent) consumers obtain an advertisement directly, so that they can bind the new peer service directly.

Apart from the modification to the peer service specification (modification of the '*C*' structures), all tailoring steps are also delegated to the running instances of that service (delegation to '*P*' proxies which then apply each step to the pertaining FLEXIBEAN components). All interface and local compositions are then tailored promptly. It is assumed that these changes do not result in unpredictable or wrong be-

[22] An acknowledge or disacknowledge message can be sent through the DEEVOLVE MailClient

[23] The interested reader should refer to http://www.sascha-alda.com/deevolve to read the API doc of DEEVOLVE. Here, more information on the implementation is given.

havior at consumer side. This can be justified by the imposed adaptation policies. All consumers are supposed to have enough time to prepare themselves to the announced changes. More information on the DEEVOLVE's tailoring service and on the interplay among the involved objects during the tailoring process can be found in section 8.4.4.

7.6 Related Work and Scope

The management of consumer dependencies within a service-oriented architecture for supporting service adaptation is hardly addressed in scientific literature. Consequently, there is no fundamental reference work that one could compare with the presented material of this work. The following related works tackle dependency management in different ways.

Available contributions on dependency management in service-oriented architectures merely aim at describing inter service dependencies at abstract levels. In the work of [Verma *et al.*, 2004] the authors describe a way for accommodating inter service dependencies in abstract specifications of business flows (BPEL4WS compositions) using ontologies. Their work promises to improve the discovery of new services for a given abstract specification. They focus on scenarios, in which the selection of a new service may depend on the selection of other services. After having set up an ontology relating dependent services, they use an inference method to verify if a newly located service satisfies the dependency requirements with respect to that ontology. The same method is able to identify all services in a business flow that need to be updated after the selection of a service. Verma et al. use DAML, a standard ontology markup language for expressing dependencies. More ontology-based approaches for describing inter-dependencies in SOA can be found in the related work section of Verma (cf. [Verma *et al.*, 2004], section 5). A similar approach but with a proprietary ontology (dependency markup language (DAM)) can be found in [Tolksdorf, 2003].

To some extend, the ontology-based model and the inference mechanism such as of Verma et al. could be used to identify consumer dependencies to local services in the scenario of a service-oriented peer-to-peer architectures. The approach in this dissertation is, however, a bit more flexible. An ontology is determined usually during design time and remains fix later during use time. New (types of) service, consumer, and dependencies cannot be included in a fixed ontology. The presented approach is *open*, that is, any peer is able to register arbitrary services into a peer store of a provider. Verma's approach could also be used to consider dependency values expressing the relevance to a service (attribute field in an ontology). Again, the resulting model would be fix. DEEVOLVE accomplishes the extension of the set of available conditions (see section 7.4.4). In principle, all investigated works of this area envision the *autonomous* modification of service composition specification (i.e. discovery and integration of services) with as little human intervention as possible. The DEEVOLVE approach (and the underlying $SO_{P2P}A$ model) stress human participation as important. This work illustrates adaptation policies to take into account the perspective of end-users during the modification of a service composition. Such a user perspective is ignored in either of the inspected related works.

A number of related works can be found for dependency management in the area of component-based distributed systems. A fundamental and highly cited work originates from the two authors Kon and Campbell [Kon and Campbell, 2000]. The authors propose a generic model to reify dependencies in distributed component systems. This model promises to make it possible to develop dynamically modifiable component-

based systems. They distinguish between two types of dependencies, prerequisites and dynamic dependencies. Prerequisites correspond to requirements that must be fulfilled at start time to load and execute a component (e.g. the nature of hardware, the services it requires). Dynamic dependencies conform to actual runtime dependencies between loaded components in a running system. These runtimes dependencies are stored for each available component in a so-called *component configurator*. Each configurator is capable of storing *hooks* (i.e. components on which a given component depends) and *clients* (i.e. components that depend on a given component). A component's configurator can now be used to announce the adaptation of a component (here: the deletion) to any dependent component. This gives the components the opportunity to delete them or to reconfigure them in some way to deal with the loss of that dependency. A prototypical implementation of an architecture including the mechanism is based on the CORBA Component model CCM in connection with the TAO ORB.

The work of Kon is closely related to the approach outlined here. Consuming components are able of subscribing to a provider component that notifies them on adaptation actions. However, announcements are sent only at the time when the adaptation has been made. This is clearly not sufficient for critical applications. Anyhow, the authors reason about this problem and suggest a *policy* in order to regulate the adaptation of a component. The two policies they propose accord to the policies of "Liberal" and "Conservative" adaptation from this work. Although they foreshadow the premise of policies that "lie somewhere between these two extremes", no concrete information is provided on these. This works fills this mentioned gap by two additional policies, "Notification" and "Negotiation" before adaptation. In particular the policy requiring the negotiation with available consumers fosters the involvement of users in the adaptation process. The user perspective is completely out of focus in Kon's work.

Two further works for managing dependencies in distributed applications are worthwhile presenting. Hasselmeyer presents a JINI-based software architecture that allows for storing and visualizing functional dependencies among clients and their dependent service providers [Hasselmeyer, 2001]. Like in this work, he proposes a graph-based visualization and, on top of that, graph-based analysis rules. However, only the client's perspective is considered, that is, component on which a client depends on. Therefore, no adaptation policies (including conditions and strategies) are included. In the work of Keller and Kar, the authors explain an extensive catalogue of how dependencies between components can be described [Keller and Kar, 2000]. Dependencies are also stored outside the components in extra files that are stored within a database. Offline, graph-based analyses can be run on that data. Adaptation strategies that incorporate , for instance, the notification of other networks node are not considered.

Chapter 8
Adaptation Methods for Runtime Exception Handling

This chapter presents additional concepts of DEEVOLVE for handling exceptional cases that occur due to the unavailability of single peers. The $SO_{P2P}A$ architectural style distinguishes between two phases for handling exceptions, namely exception *detection* and exception *resolution*. During exception detection, a peer environment monitors its dependent peers and maintains their corresponding status (e.g. alive, failed). Given the occurrence of an exception (i.e. the failure of a peer), pre-defined *options* can be determined that execute *actions* to resolve that exception (e.g. the discovery of an alternate service or the binding of a new service). At selected decision points, an end-user can be involved for selecting suitable options or for carrying out own actions. Basically, actions conform to *component-based adaptation methods* that adapt a service composition at its *compositional level*. Actions also embrace behavior for notifying users, discovering new services or for opening tools. The benefit of this approach is that it relieves services from incorporating complex code for handling exceptions.

This chapter merely focuses on exceptions that might lead to the violation of integrity constraints imposed on several distributed, interacting peers. Peers (i.e. their respective owners) are capable of establishing such architecture-wide constraints to formulate contracts between participating peers. These contracts entail both obligations and benefits for a peer in terms of the availability and reliability of their consumed and their provided peers. Integrity constraints are presented as an important contribution in section 8.2. Following that section, the phases of exception detection and handling are elaborated in section 8.3 and 8.4, respectively. At first, an overview of the whole exception handling mechanism featured by DEEVOLVE is outlined in section 8.1.

8.1 Overview on Exception Handling in DEEVOLVE

As pointed out in the beginning of this chapter, human end-users are an important actor during exception handling in DEEVOLVE. The allocation of relevant activities to actors (user and system) during exception handling is pictured in Figure 8-1. All activities can be assigned to two principle phases, that is, design and use time. During design time, a *component assembler* is responsible for instrumenting a (given) PeerCAT-based service composition with additional integrity constraints and exception handlers. A component assembler has the overall competence to understand the service composition. He decides which services should be part of an integrity constraint in order to increase the reliability of the entire application. In addition, he determines

which exception handlers fit best to the context of the service composition. The component assembler is not necessarily the same person as a service provider. The service provider could ask an external expert to assemble his services so that the provider can use the resulting composition internally or publish it as a peer service to other peers.

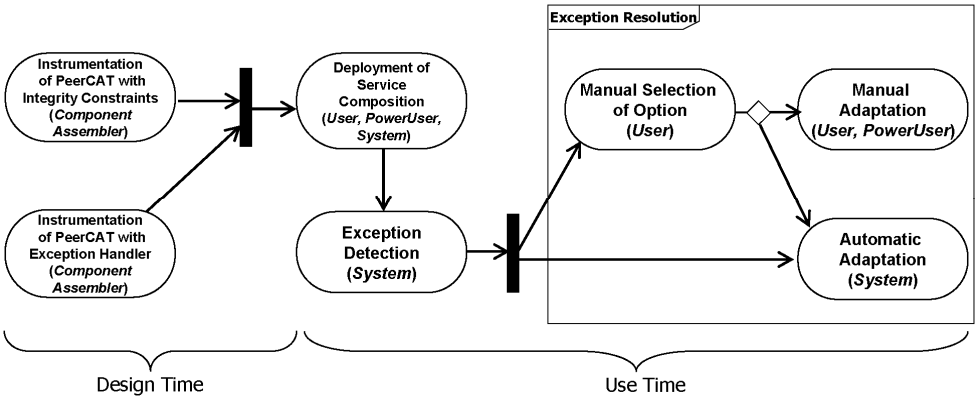


Figure 8-1: DEEVOLVE's phase model for exception handling

After having finished the instrumentation of the PeerCAT files, a *user* is responsible to initiate the deployment of the service composition. This could either be a service consumer who has retrieved the service advertisement and makes use of the peer service. Else, this could be the local operator of a DEEVOLVE peer environment acting as a local consumer. Besides the deployment of the actual peer services, DEEVOLVE parses the (declarative) descriptions of the exception handlers as well as of the integrity constraints and transforms these into concrete, executable (Java) code.

After the deployment, the *system* (i.e. the DEEVOLVE environment of the user) starts monitoring dependent peers. These mechanisms are capable of detecting the sudden unavailability of any dependent peer. Having detected such an exception, DEEVOLVE can execute two types of activities to resolve or to handle the exception. Firstly, the system can carry out actions *automatically*. These actions have been predetermined in an exception handler during the instrumentation of the respective PeerCAT file of the service composition. Alternatively, the user is capable of *manually* selecting options out of a set of pre-defined options to handle the exception. These options can either trigger actions implementing adaptive behavior (e.g. a procedure to locate and to bind a service) or enable users to pursue own adaptation routines manually. This option is suitable for more sophisticated users, who are experienced in tailoring component-based software. Such a user is indicated as power user. As shown later on, both automatic and manual adaptation can be mixed within one handler.

A user is always able to use both the predefined exception handlers and integrity constraints directly. Power users are capable of overriding these by their own handlers and constraints. This approach allows them to deploy their individual handlers and constraints according to local restrictions or requirements. The substitution of existing default handlers occurs prior to the deployment of a service composition (in Figure 8-1 indicated by activity "Deployment of Service Composition"). Within the next sections, the relevant activities will be elaborated in more detail.

8.2 Integrity Constraints

This section illustrates the notion of integrity constraints. These constraints are used to formulate additional conditions on service compositions as well as on a peer-to-peer

architecture. At first, the principle ideas are described. Following that, two concrete examples for integrity constraints are highlighted.

8.2.1 Principle Idea

The $SO_{P2P}A$ architectural style suggests integrity constraints as an option to define explicit *conditions* that need to be fulfilled during runtime of a service composition. Such conditions could necessarily address all properties of the constituting elements of a service composition (i.e. peer services, components, ports, bindings). An integrity constraint enriches a given service composition with additional conditions. For the special requirements of a dynamic peer-to-peer architecture, however, specific integrity constraints are necessary to describe conditions in terms of the availability of peer services within a given service composition. Such a constraint could be useful in an application *context*, in which the availability of single peer services is mandatory. This way, one could define the core functionality of a service composition that must always be available during a given (working) context. A constraint to define such a core composition or *minimal composition* is introduced in section 8.2.3 as an example for an integrity constraint. Further constraints are useful to describe conditions on transitively dependent services. Such conditions can be applied in scenarios, where peers carry out sequential activities (see scenario in section 9.5 for a striking example). To define such constraints, the information flow integrity will be introduced in section 8.2.4.

With respect to $SO_{P2P}A$, peer operators (or service provider) can adopt the notion of integrity constraints as an opportunity to define *service contracts* among each other. Especially in collaborative scenarios, where peers act cooperate for reaching a common goal (see scenarios in section 9.5) such service contracts could be beneficial to guarantee the performance of single activities. A service contract could, for instance, be formulated for peers within a given group. The negotiation process and the final agreement of a service contract is not covered in this work. It is assumed that peer operators have agreed on specific service contracts during the set-up of a peer group.

If a condition is violated, a dedicated handler is invoked to resolve exceptional cases. Handlers are pre-determined during design time by a component assembler. These handlers, however, do not entail a fixed way for handling the exception, that is, for restoring the condition. This is in particular not feasible in application scenarios, where the context in which a service composition is operating cannot be estimated accurately in advance. The same problem occurs in scenarios, in which the context is simply too complex to be described (e.g. a context that should take into account project measures or managerial aspects). The approach promoted in this work is as follows: the component assembler provides a couple of possible exception handlers during design time. During runtime (i.e. when the exception occurs), a dedicated user can select a handler he believes to be the most appropriate one for a given application context. An example for selecting a handler according to various contexts will be described in section 9.5.

8.2.2 Definition of Integrity Constraints in PeerCAT

The structural model of a peer service composition (Figure 6-12) can be extended to incorporate the elements of an integrity constraint. This new structural model is depicted in Figure 8-2. A PeerCAT-based service composition can reference an arbitrary number of different (integrity) constraints. A constraint associates many *contexts* that

entail, *when* a given constraint is valid. The user can choose and switch among many contexts at runtime. Given a selected context, all constraints that are associated to it must be met in all instances of compositions pointing to these constraints.

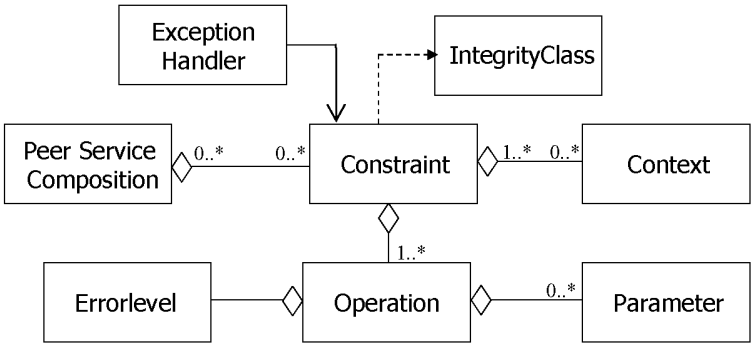


Figure 8-2: Structural model of service composition including integrity constraints

A constraint is furthermore characterized by a number of *operations*. An operation represents a condition that must be fulfilled in the context of the associated composition. A condition may take additional *parameters* that represent target values. An *errorlevel* dictates the relevance of a condition, which itself is checked periodically by the DEEVOLVE environment. Although many reasons for triggering an integrity check could be reasonable (e.g. user-triggered, time-triggered), an integrity is only checked whenever an exception has occurred (i.e. the failure of a peer). If the condition is not fulfilled, the constraint is said as violated. The actual algorithm for checking the correctness of an integrity constraint is implemented in the "IntegrityClass" class. Each defined integrity constraint must consist of such a class. Within an algorithm, the expected target values of an operation are compared against actual values that are passed to an object of class "IntegrityClass". Such object must implement a special interface "IntegrityInterface" to become a valid integrity constraint within DeEvolve (see [Palij, 2006] for more information).

```
<composition name = "aName" ID = "anID" > ...
   <services> .... </services>
   <bindings> .... </bindings>
   <exceptionHandling> .... </exceptionHandling>
   <semantics>
     <constraint name = „“ ID = „“ >
        <description> ... </description>
        <integrityclass classname = „...“ />
        <operation = „aOperation“ > // see table 8-1
            <params = „ ...“ />
            <errorlevel value = „ “ description = “” />
        </operation>
        <context value = „ “ />
     </constraint >
   </semantics>
   <dependencies> .... </dependencies>
<composition>
```

Figure 8-3: Template for specifying an integrity constraint in PeerCAT

An integrity constraint is described in a declarative way and included in a textual PeerCAT service composition. The actual declaration is enclosed within the tags <Semantics> … </Semantics> (see Figure 8-3). Within these tags, the declaration can refer to all constituting elements that have been defined in the corresponding PeerCAT description. Recall that the declaration does not contain any code for *verifying* the cor-

rectness of an integrity constraint. The declaration of the operation (tag <operation>) is the most important statement. The component assembler can refer to a couple of predefined operations (see Table 8-1). These operations are bound to a *condition level* that entails the scope in which an operation is valid. Although many more levels down to a fine-grained level are supposable (i.e. level of single components or ports), only three condition levels are proposed here, *architectural*, *peer group*, and *service composition* level. Operations on an architectural level describe conditions that are valid for the entire peer-to-peer architecture. Here, conditions can be formulated that state, which type of consumer peers or peer services should be connected with a local peer service that has been formulated as a peer service composition. Also, transitive dependencies along many peers can be formulated.

Condition Level	Operator	Parameters	Description
Architecture	hasConsumer	[peer-id], [peerService-id] or [peerRole]	Peer service has a direct consumer
	isConnectedTo	[peer-id], [peerService-id] or [peerRole]	Peer service has a direct or transitive consumer
PeerGroup	serviceHasMembership	[peerService-id], [peerGroup-id*]	Peer service must be in peer groups
Service Composition	hasService	[service-id*]	List of services that must be available in service composition
	hasBinding	[service-id_provided, service-id_required]	Services have concrete binding

Table 8-1: Overview on operations and their levels that can be applied within the definition of an integrity constraint

The single operation on a peer group level allows associating a permanent group membership to a peer service. Such a constraint is violated, if the implied group membership has been cancelled. Operations on an architectural level accomplish the definition of conditions that are valid within a single service composition. These operations allow defining conditions relating to the dependency on provided services. The formulation of integrity constraints based on the operation (conditions) and corresponding parameters conforms with the formalized presentation of integrity constraints in the $SO_{P2P}A$ architectural style (section 4.2). In the following two sections, two examples for integrity constraints are described in more detail.

8.2.3 Example 1: Minimal Composition Integrity

Owing to the dynamical nature of a service-oriented peer-to-peer architecture, an application once composed out of different peer services cannot permanently rely on the availability of each involved service. During deployment of a distributed application, the underlying component architecture establishes connections to each peer service residing on remote peers. This process of starting up an application might take an unpredictably long time, if certain services cannot be resolved immediately. Obviously,

the entire application is ready to start, if all necessary peer services have been resolved and integrated. However, in some application scenarios it is throughout conceivable to indicate an application as executable, if a *minimal composition* of peer services has been resolved and integrated. A minimal composition allows component assemblers to utilize the application in a minimal fashion. The same assumption holds during use time of a service-oriented application. Again, an integrity constraint describing the minimal application could serve as a contract among all involved providers and consumers. This contract ensures the execution of certain functionality during deployment and during runtime of the application. This is especially important in collaborative settings, in which peers collaborate in a *synchronous* and *parallel* way towards a common working goal.

```
<composition name = "Extended Groupware" ID = "word" >
 <services>
  <service name = "Word Processor"  ID = "word" host = "localPeer" >
  <service name = "EmailService"  ID = "mail" host = "peer1" >
  <service name = "PrintService"  ID = "print" host = "peer2" >
  ...
  <semantics>
   <constraint name = „Minimal_Composition" ID = „activity1" >
        <description> This constraint describes a minimal groupware application </description>
        <integrityclass classname = „org.deevolve.integrity.MinimalComposition" />
        <operation value = „hasService" >
            <param key = „service" value = „word" />
            <param key = „service" value = „mail"  />
            <errorlevel value = „obligation" description = "All services must be available" />
        </operation>
        <context value = „Daily use of the groupware application" />
   </constraint >
  </semantics>
</composition>
```

Figure 8-4: PeerCAT file representing a minimal integrity constraint for a Groupware application

The facility temporarily doing without a peer service depends on the intended role of a peer service within the composition. Consider for example the "Extended Groupware" composition as illustrated in Figure 8-4. As a reliable and efficient email connection has become an indispensable tool for many work processes, the availability of the "EmailClient" service offered by peer A is without any doubt a crucial requirement. Therefore, an integrity constraint of type "Minimal_Composition" as depicted in Figure 8-4 is proposed to describe a minimal composition consisting of the local service "Word Processor" and the remote service "EmailClient". Whenever both the "Word Processor" service and the "EmailClient" service can be integrated during deployment and are constantly available during runtime of the composition, the integrity constraint is assumed as fulfilled. The integrity constraint will be checked, if any depended peer from the local peer becomes unavailable. If the unavailable peer does indeed host one of the services defined within the minimal composition, the exception crops up the violation of an integrity constraint. The local DEEVOLVE environment then, in turn, throws an exception and invokes the corresponding exception handling routines (see section 8.4). Note that the failure of the peer service "PrintService" does not effect the violation of the minimal composition integrity constraint.

The integrity constraint defined in Figure 8-4 applies in the context "Daily use of the groupware application". This context implies that the integrity constraint should be valid during all use times. Apparently, the component assembler (or user) is able to define additional integrity constraints that apply at, for instance, special working con-

texts (e.g. “Night use”, “User is absent”). The user is able to dynamically switch between various integrity constraints depending on the actual working context within the DEEVOLVE console.

8.2.4 Example 2: Information Flow Integrity

Groupware applications aim at supporting the work of multiple users with different roles and responsibilities in a collaborative manner. In many working scenarios, these users often perform activities according to a given workflow model. Typical workflow models (e.g. Petri nets or UML activity diagrams) allow for modelling parallel and sequential working activities (including operations for splitting and joining activities). In this light, not only the integrity of direct connections between services is of importance, but also the integrity of the whole workflow and the information flow itself.

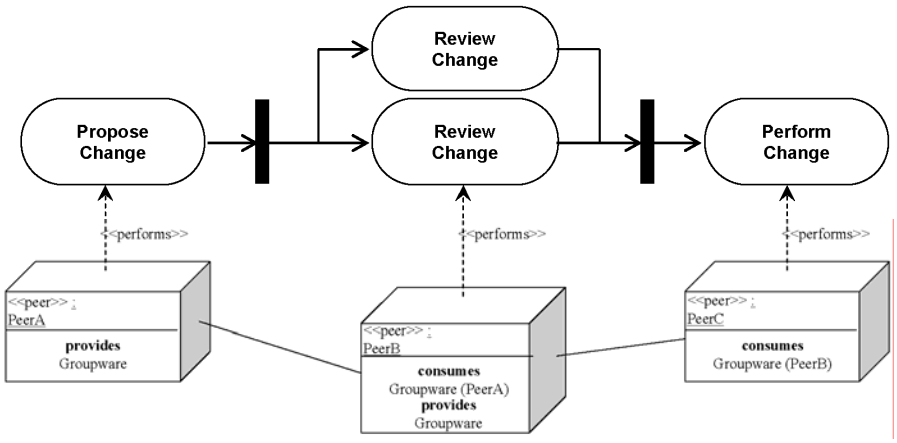


Figure 8-5: A workflow described by an UML activity diagram. The activities are mapped to concrete peers

Consider for instance the example composition depicted in Figure 8-5, which realizes a simple workflow. The activities in this workflow are assigned to concrete peers, which perform these activities in the local application “Groupware” (e.g. by editing or reading a text in the word processor). These peers are supposed to use remote capabilities of “Groupware” in order to activate further activities on peers that have subscribed as a consumer to published “Groupware” peer service. Again, an integrity constraint is formulated only within a local peer environment. If each peer environment describes a constraint, then a global constraint is available for the entire peer-to-peer architecture. An integrity constraint for peer A, for instance, could dictate that peer C must be available before, during, and after the (parallel) review activities. This integrity constraint is given in Figure 8-6.

The information flow integrity of Figure 8-6 makes use of the “isConnectedTo” operation. The associated parameters say that peer (1st argument) “PeerC” (2nd argument) must be connected with the given peer service “Extended Groupware”. This can either be a direct or transitive dependency. If peer C becomes unavailable, say, during the review activities, peer B would recognize this exception immediately. DEEVOLVE’S ability to cascade exceptions to other peers (see section 8.3.5) facilitates peer B to delegate the exception to any of its dependent peers. In the case of Figure 8-6, peer A would be notified as well. Given the unavailability of peer C, peer A can also throw an exception to announce the violation of the integrity constraint. Again, the DEEVOLVE

environment of peer A then could execute corresponding exception handlers to resolve the exception.

```
<composition name = "Extended Groupware" ID = "word" > //PeerCAT of PeerA
  ...
  <semantics>
   <constraint name = „InformationFlow_Composition“ ID = „ activity" >
        <description> Ensures Connectivity in a Workflow </description>
        <integrityclass classname = „org.deevolve.integrity.InformationFlow“ />
        <operation value = „isConnectedTo “ >
            <param key = „peerType“ value =  „peer“/>
            <param key = „peerID“ value = PeerC“ />
            <errorlevel value = „obligation“ description = “Activities on Peer C must be available” />
        </operation>
        <context value = „Reviewing changes: Peer C must be available at all times“ />
   </constraint >
  </semantics>
</composition>
```

Figure 8-6: Information flow integrity defined in PeerCAT

The information flow integrity can also accept a role name as a parameter (1st argument = “role”). The integrity constraint could point out that a peer with dedicated semantic role has to be connected directly or transitively to the given peer service. With respect to the scenario of Figure 8-6, peer C could take over the role “Performer” denoting its responsibility to perform changes after the review activities. The constraint of peer A then could dictate that peer service “Groupware” always has to be connected with a peer possessing the role “Performer”. The usage of semantic roles presumes that an information concerning semantic roles of a peer have been exchanged during the subscription phase (cf. section 7.2.1).

8.2.5 The Deployment of Integrity Constraints

Integrity constraints are deployed during the actual deployment of a service composition within the service consumer environment. DEEVOLVE parses the declarative description of a constraint and turns it into an object-based representation (i.e. Java objects) with respect to the structural model of Figure 8-2. An object model is stored internally in the DEEVOLVE environment. An integrity can be addressed through DEEVOLVE’s integrity checker service (see section 6.6.1), whereas a check is pursued after the detection of an exception. The checking procedures are explained separately in section 8.3.4 after the introduction of the detection mechanisms in the next section.

There is no upper limit on the numbers of usable integrity constraints. A component assembler can always introduce new constraints on demand. New constraints including the integrity class can be plugged in dynamically. DEEVOLVE facilitates this extensibility feature by implementing the strategy pattern [Gamma *et al.*, 1995]. The system treats each integrity class as a concrete strategy object that has to implement methods provided by an abstract strategy class. The integrity checking service accesses an integrity object (=strategy) through a decoupled interface (the context) that is independent from any concrete strategy object. Depending on the currently valid integrity constraint, an integrity selector class (= *policy*) binds the pertaining concrete integrity object to the interface. DEEVOLVE peer environments not possessing integrity classes are able to get these from other peers via DEEVOLVE’s class loading mechanism.

8.3 Exception Detection

This section describes the models implemented in DEEVOLVE for detecting an exception. As already pointed out, an exception denotes the unavailability of a dependent peer or peer service. More exception types (e.g. the violated of a peer service's state, the loss of a peer group membership) are necessarily conceivable and realizable. This work, however, only focuses on this single exception type.

It turned out that two different models are essential to facilitate reliable exception detection. The first method realizes a monitoring approach for directly tracing the availability of dependent peers. The second approach introduces the notion of a broker responsible to delegate calls on provided ports of an available peer service. The broker is also able to catch possible runtime exceptions and to delegate them to the DEEVOLVE environment. Both approaches for exception detection have been formalized in $SO_{P2P}A$ (section 3.3.11). This section presents the implementation of these in some detail. Much more information on the implementation of both concepts can be studied from the master thesis of Alekseij Palij [Palij, 2006].

8.3.1 Monitoring of Peers

The monitoring concept for observing dependent peers is implemented in DEEVOLVE'S monitoring service residing on its core layer (see Figure 6-18). This monitoring service works according to the following principle: If a peer subscribes as a consumer for a provided peer service in the DEEVOLVE environment of a service provider, then the service providing peer will be asked to send ping signals to that service consuming peer in regular time intervals. If the consumer peer receives such signal, then the dependent provider peer is said to be alive. If no signal arrives after a fixed period (e.g. 10000 milliseconds), the dependent peer is supposed as failed. The status of a peer is stored in the "PeerStore" class of a DEEVOLVE environment (see section 7.2.2, Figure 7-4). After the omission of a ping signal, attribute "status" of the respective "PeerService" and "Peer" entry is set to "failed". Whenever a failure of a peer occurs, the affected dependent peer services are computed. Based on this set of services, all deployed instances of service compositions are derived that use at least one of those failed peer services. If any of these compositions have deployed integrity constraints in their PeerCAT files, these integrity constraints are checked by using the integrity checker service (section 8.3.4).

In accordance to the $SO_{P2P}A$ formalism, a consumer peer can also send an "alive signal" back to the provider upon reception of the provider's signal. This enables a provider peer to perform the validation of specific integrity constraints such as the information flow integrity (section 8.2.4) that incorporate the state of service consumers in their internal condition. A provider peer has to enable the reception of status signals from a consumer by setting the "cascade" attribute to value "yes" during the subscription process (see section 7.2.1).

The implementation of the monitoring concepts uses the novel ping concept of Java 5.0. It allows Java programs to send ping messages to a specific network host having a concrete IP-address. For more information on this technique, see [Flanagan, 2005].

The monitoring approach is certainly practical but in some cases not very effective. Consider the following use case: a provider sends an "alive" signal to its dependent consumer peers. Another signal of that peer is expected a couple of minutes later. If the provider peer, however, already fails during these two time points, the (remote)

interaction with the affected peer service would potentially lead to a failure in the local service composition (see [Palij, 2006], section 3.2.1 for a good illustration of this problem). In order to avoid such an exceptional case, the *broker-based remote interaction* model is proposed. This second model will be outlined in the next section.

8.3.2 Broker-based Remote Interaction

The broker-based approach for remote interaction aims at suppressing potential misbehavior caused by the failure of a provider peer hosting a dependent service. Recall that a peer service consists of a local and an interface part (section 6.3.3, Figure 6-7). The interface part of a peer service is migrated to the consumer peer, while the local part (i.e. the actual implementation of the peer service) remains at the provider side. The purpose of the interface part is to accept any kind of input from the consumer (i.e. through a GUI component or by another peer service) and delegate this to the remote interface part. Obviously, this remote interaction between interface and local part fails, whenever the provider hosting the local part is unavailable.

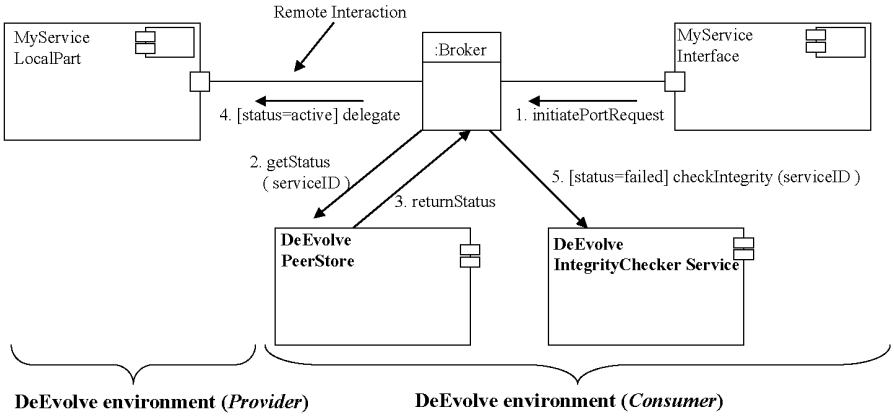


Figure 8-7: Broker-based remote interaction to delegate port calls between the interface and the local part of a peer service in DEEVOLVE

The $SO_{P2P}A$ style proposes a *Broker* process in order to control the remote interaction between interface and local part (section 3.3.11). This broker only permits an interaction between interface and local part, if the respective status flag of the hosting peer (process) is set to "active" (status flags are collected by the *Controllerprocess*). DEEVOLVE adopts this approach. Whenever an interaction is to be performed between ports of a local and a remote part, a broker object is placed "between" these two ports (Figure 8-7). The purpose of this object is to *delegate* between these two ports. During the delegation, the broker queries the "PeerStore" object concerning the current status of the remote peer. The remote interaction is authorized, if the status of that peer is "active". If the "PeerStore" object returns "failed", the delegation will be blocked. In order to start resolving the exception, the broker then first calls the integrity service of DEEVOLVE (see next section). This service can later call the handler service of DeEvolve to handle the occurrence of that exception. This approach realizes an exception handling *on demand*: an exception handling is only pursued, if a user makes use of the affected service composition.

The broker-based approach is also capable of detecting exceptions. As described in the previous section, status information taken from the "PeerStore" class is definitely not reliable, especially when a huge interval for emitting ping signal is chosen. A pro-

vider could be unavailable though it has an "active" status. In this case, the broker would try to delegate the port call the local part of the peer services. From a technical perspective, a remote method as declared in the Java interface for that provided port is invoked by using the RMI technique. If the provider peer and, thus, the method is unavailable, an exception (`java.rmi.RemoteException`) is thrown in the broker. The broker then branches to the catch clause of the corresponding try-catch block that surrounds the method invocation. Within this catch block, the broker updates the status of the corresponding remote peer service and of the hosting peer. Following that, the integrity checker service is invoked to check the potential violation of integrity constraints (see section 8.3.4). Again, DEEVOLVE can potentially invoke the handler service later on, if the integrity constraint has been violated due to that exception. The update process of the broker is visualized in Figure 8-8.

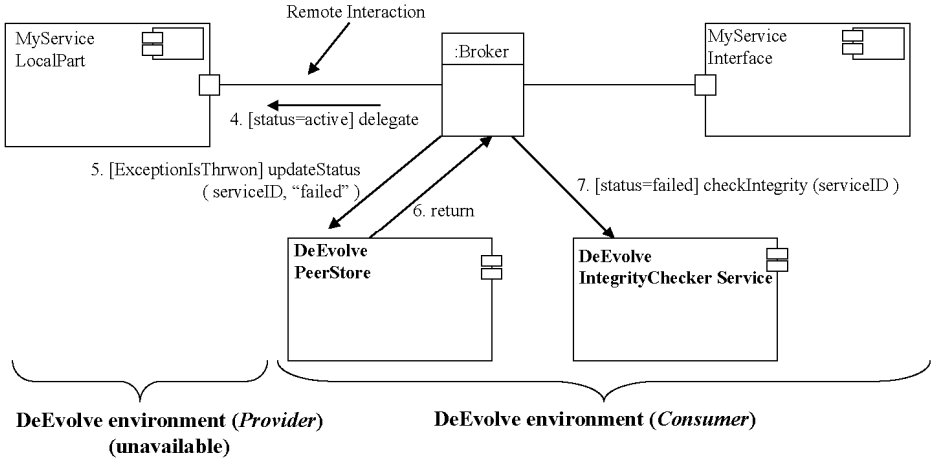


Figure 8-8: Scenario for a broker-based approach when an exception is thrown upon remote method invocation

Technical Realization

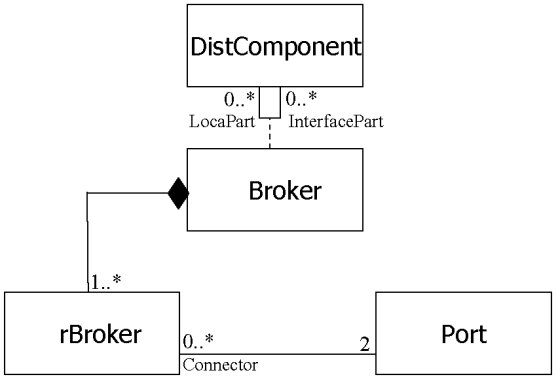


Figure 8-9: Partial structural model of DEEVOLVE encompassing broker approach

A broker object is always necessary between the interaction of a required port (consumer side) and a provided port (provider side). According to the structural model of DEEVOLVE introduced in section 6.3.4, these broker objects reside between the DistComponent objects representing the local and the interface part of a peer service (cf. Figure 6-8). The structural model encompassing the broker-based interaction can be refined as depicted in Figure 8-9. Class Broker is a composite holding all concrete brokers (rBroker) between ports being part of a local or interface part (DistComponent). An object of class rBroker connects exactly two objects of class Port.

It has been a requirement during the implementation of the broker approach that existing peer services or FREEVOLVE applications can also benefit from this approach without cumbersomely modifying them. All broker objects are therefore automatically generated based on the given CAT-XML description of a peer service as well as on the class definitions from the constituting FLEXIBEAN components. Brokers are injected during deployment of a peer service within the consumer's peer environment. Whenever the deployment algorithm detects (by inspecting the CAT-XML description) an interaction between a required port from an interface part and a provided port from a local part, a broker is created. The required port is then passed the reference to the broker object instead of the FLEXIBEAN component holding the provided port. The broker object provides the same interface as the one of the provided port. The broker's interface is arranged by analyzing the Java interfaces of the respective provided port using Java's reflection technology. So, from the perspective of the required port, the interaction is completely transparent. The broker implements further (hidden) code for addressing the local "PeerStore" and for invoking the integrity checker service. If the status check from the peer hosting the local part of a peer service is positive, the broker delegates the call from the required port to the provided port. More details on the implementation of the broker approach can be read in [Palij, 2006].

8.3.3 Evaluation of Broker Generation

The creation of a broker includes the compilation of Java code during runtime. Evaluation results have shown a critical slow down of the deployment process ([Palij, 2006], section 6.3). However, all brokers only need to be created once, that is, during the first deployment of a peer service. The broker classes are then stored locally within the consumer's DEEVOLVE environment. For the second and further deployment of the same service, thus, these pre-compiled brokers are used, making the deployment process more efficient. For the time being, brokers are compiled *sequentially*, which is certainly critical when many remote port interactions are given. A speed up could be reached, if the compile process is organized as a parallel process, in which the brokers are compiled concurrently.

8.3.4 Verifying Integrity Constraints

Each time a broker object or the "PeerStore" itself detect an exception, both call DEEVOLVE's integrity checker service to see, whether the exception violates an integrity constraint of a currently deployed and running service composition. This call always passes the identifier of the lost service to the service, which is stored in broker as a private attribute during deployment. Figure 8-10 visualizes the interaction between a broker and the integrity check service.

After receiving the call, the service figures out, which service compositions are affected by that lost service, and the currently selected working context is identified. Given both the set of PeerCAT-based service compositions as well as the context, the integrity service is able to derive the currently valid integrity constraints. Afterwards, all integrity classes are invoked by calling method "evaluateIntegrity". Each integrity class must provide this method. An integrity class has to implement the Java interface org.deevolve.integrity.IntegrityStrategy to ensure that this method is given. An object storing information about the lost service as well as an object describing the current working context is conveyed to the integrity class. Internally, the integrity class

can compare these values (representing actual values) with the pre-declared values from the PeerCAT structure (representing the target values). If the integrity is indeed violated, then integrity checker service calls DEEVOLVE's handler service to invoke the adequate handler being associated with that integrity. This procedure will be described in greater length in section 8.4.4.

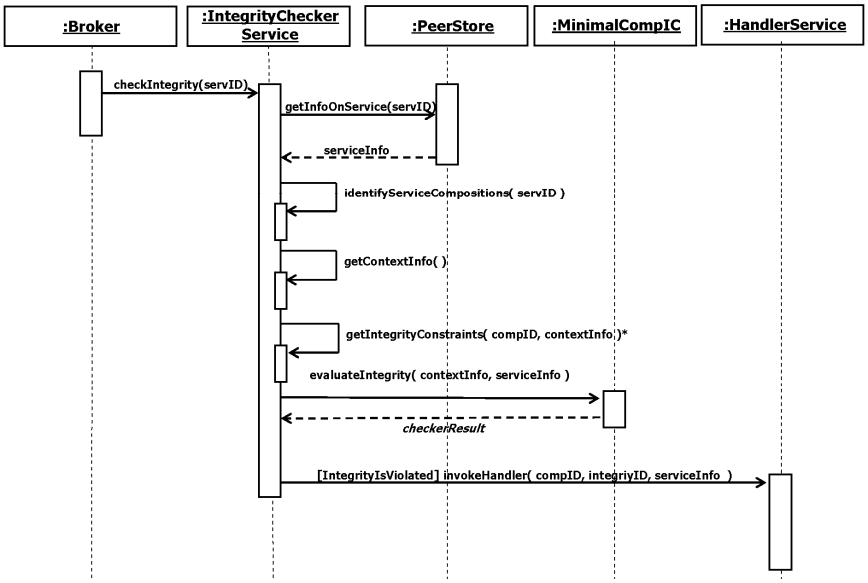


Figure 8-10: Sequence diagram to visualize how an integrity is checked

8.3.5 Exception Cascading

All DEEVOLVE peers are able to register in other DEEVOLVE environments to be notified upon the occurrence of an exception. This way, an exception cannot only be detected within a local peer environment but also be cascaded to other peers, as well. The common way to register for exception cascading is to set the "cascade" attribute to "yes" during the subscription phase between provider and consumer. Whenever the unavailability of a peer is identified, the registered (dependent) peers receive a message through the DEEVOLVE message service. This message contains the id of the missing peer and the type of exception. The message service delegates this message to the monitoring service acting as a listener for messages of that type in the message service. The monitoring service itself delegates the information to the integrity checker service that puts in effect the same behavior as depicted in Figure 8-10. The derivation to the normal sequence is that calling method "identifyServiceComposition" returns null, because the lost transitive service is no direct part of a local service composition. The constraints are then computed only based on the current context and evaluated accordingly. If an integrity constraint has been violated, all associated handlers part of currently running service compositions are invoked by calling the handler service.

Exception cascading is useful to handle the violation of architecture-wide integrity constrains such as the information flow integrity (see section 8.2.4).

8.4 Exception Resolution

Exception resolution refers to the phase of resolving an occurred exception. During this phase, an end-user is mainly involved. Among a set of predefined exception han-

dlers, he is capable of selecting the most appropriate handler being suitable for a given application context. This section describes exception handlers in more detail.

8.4.1 Definition of Exception Handlers in PeerCAT

In analogy to the notion of integrity constraints, the notion of exception handlers extends the structural model of a service composition as described in section 6.4.2 (Figure 6-12). This refined structural model embracing the elements of exception handlers is depicted in Figure 8-11.

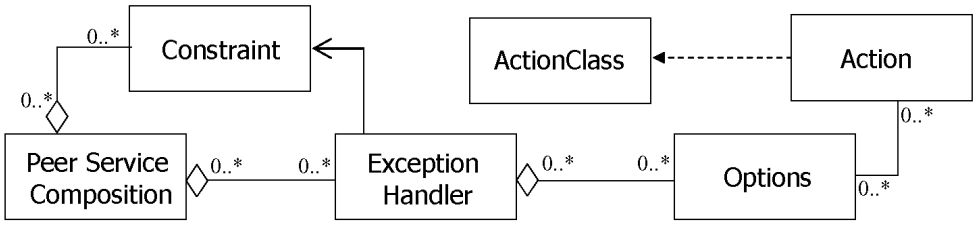


Figure 8-11: Structural model of a service composition with exception handlers

A PeerCAT-based service composition may potentially refer many *exception handlers*. Each exception handler associates one *integrity constraint* that is defined and valid within the associated service composition. The exception handler itself is capable of offering many *options* that feature some behavior for resolving an exception. Options make use of *actions* that represent core functionality for resolving an exception (e.g. binding ports, discovering a service, notifying a user). Options concretize actions by adding private parameters to them. The actual behaviour of an action is implemented in the corresponding *action class*. Like the integrity constraint approach, further actions can be added to the description, thus, there is no restriction on the number of available actions. An action has to implement a Java interface in order to become a DEEVOLVE-compliant action deployable in an exception handler (see section 8.4.4).

```
<composition name = "aName" ID = "anID" > ...
  <services> .... </services>
  <bindings> .... </bindings>
  <exceptionHandling>
    <actions>
      <action id ="actionName" class="aClass" description = "desc" />..
    </actions>
    <handlers>
      <handler integrity = "integrityID" >
        <options>
          <option id = "optionID" action = "actionID">
            <param name = "aName" value = "aValue"> ...
          </option> ...
        </options>
        <automatically>
          <run option = "optionID" description = "Description of option"/>
        </automatically>
        <manually>
          <select option = "optionID" description = "Description of option" />
          <select option = "opD1, opD2" mode = "parallel | sequential" description = .. />
        </manually>
      </handler>
    </handlers>
  </exceptionHandling>
  <semantics>...</semantics>
  <dependencies> .... </dependencies>
<composition>
```

Figure 8-12: Template for specifying exception handlers in PeerCAT

Figure 8-12 shows the structure for defining exception handlers within PeerCAT. All declarations must be placed within the tags <exceptionHandling>…

</exceptionHandling>. At first, actions are specified that can later on be used by the concrete handlers. Each action has a unique id and associates a class. Code within this class can make use of different APIs such as the TailoringAPI to adapt peer services, to look for new services, or to invoke tailoring tools and so on (see section 8.4.4).

The actual handlers are declared Within tags <handlers>...</handlers>. Each handler refers to exactly one integrity constraint ("integrityID") being part of the same surrounding composition. Whenever this integrity constraint is violated, this handler will be invoked (see section 8.4.2). A handler features a set of options that simply refers to concrete actions a component assembler has defined beforehand. Here, these actions can be passed further concrete parameters. These parameters concretize actions according to individual demands of that specific handler.

The last two tags describe control elements, how the declared options are executed. Any option that should be executed automatically, that is, without the intervention of a user is declared within tags <automatically>…</automatically>. The execution of an option is indicated by tag <run option = "optionID">. If the user should be involved within the selection of an option, then all dedicated options have to be placed within tags <manually>…</manually>. The selection of an option is indicated by tag <select>. It is possible to regard options as a coherent bundle that can either be executed sequentially or concurrently. The respective execution mode is declared by attribute "mode", which could either possess the value "concurrent" for parallel or "sequential" for sequential execution of the options. The bundle of options is formulated by a comma-separated list. For the definition of exception handlers, the component assembler is able to refer to a XML-based document type definition (DTD) [Palij, 2006].

8.4.2 Adaptation Methods as Actions for Handling Exceptions

With respect to the $SO_{P2P}A$ architectural style, actions within the handler declarations merely adopt component-based adaptation methods for handling occurred exceptions. These adaptations methods have been formalized in section 4.1.1 (Table 4-1). This section describes how these adaptation methods can be applied within action in a PeerCAT-based service composition. Table 8-2 summarizes all actions that are currently supplied by the DEEVOLVE environment.

Action Level	**Action**	**Parameters (passed within options)**	**Description**
Composition	discoverService	[queryString]	Discovers a peer service with the query "queryString". Query is directly invoked. Results are presented to user and then passed to next option
	applyMembership		Applies for group membership to peer groups to which a new service many belong to
	subscribeToService	[serviceID], [dep], [cascade], [comment], [semantics]	Subscribes to the service with the necessary parameters (cf. section 7.2.1)
	deployService	[serviceID]	Deploys a peer service into a local peer environment (interface part is obtained)
	bindService	([compID], [service-id], [port-id-provided],	Binds a service into an existing service composition. Each

		[service-id], [port-id-required])*	tuple represents a binding operation between two ports of the services
	unbindService	([compID], [service-id], [port-id-provided], [service-id], [port-id-required])*	Unbinds a service from an existing service composition. Each tuple represents an unbinding operation between two ports of the services
	addService	([compID], [serviceID])	Adds a service to an existing composition. The service must be in the DEEVOLVE scope
	deleteService	([compID], [serviceID])	Deletes a service from an existing composition
User Involvement	switchToTailoring	[compID], [toolID]	Opens the PeerCAT description of the current composition into an existing tailoring tool
	showDependencies	[compID], [toolID]	Opens a dependencies analysis tool for analyzing the dependencies of that service composition to other peer services
	discoverFromServiceConsole		Opens the DEEVOLVE console enabling a user to discover a peer service manually
	discardComposition	[compID]	Discards the current composition; closes the composition
	ignoreException		Ignores the current composition
Consumer Involvement	cascadeException	[status], [text]	Cascades the violation of an integrity constraint to all subscribed peers
	notifyConsumers	[text]	Notifies dependent peers about t an occurred exception

Table 8-2: Overview on actions that can be applied in PeerCAT

Like the operators available to express integrity constraints, all actions belong to a dedicated action level. Three levels are identified, *composition*, *user*, and *consumer* level. The actions at composition level comply directly with the aspired component-based adaptation methods. These actions allow for subscribing, deploying, binding, unbinding, adding, and deleting peer services to/from a service composition. On top of these, action "discoverService" enables the location of a peer service within a peer-to-peer architecture. This is done almost without user intervention. The discovery process is started directly based on the indicated query string "queryID". The user can select upon the retrieved peer services. The chosen service is eventually passed to the subsequent option in the same handler (see section 8.4.4).

All actions defined on a user level aim at directly involving the local user to resolve the exception. Basically, three types of tools can be opened to handle the exception. Action "switchToTailoring" opens the DEEVOLVE tailoring composition tool (section 6.6.5) to tailor the affected composition in a graphical way. Action "showDependencies" opens the DEEVOLVE consumer dependency tool (section 7.4.1) in order to analyze the dependent consuming peers that might be affected by that exception. The last action "discoverFromServiceConsole" opens the DEEVOLVE console facilitating the

user to search an adequate new peer service. Besides, the user is capable of closing or discarding the running composition or simply of ignoring the exception.

The actions from the last action level aim at reporting any dependent registered peer the occurred exception. Action “cascadeException” allows to the delegate the exception of an integrity violation to all dependent peers, so that these peers can take this as an opportunity to carry out their own exception handling. The status of the peer service composition is “violated”. An information about the type of violated integrity constraint is also conveyed. Action “notify user” simply sends a message to all dependent peers. Unlike action “cascadeException”, it does not invoke any exception handling routine within the consumer’s DEEVOLVE environment. Instead, a simple message arrives in the DEEVOLVE mail client summarizing the circumstances of the exception. It does not invoke DEEVOLVE’s integrity service to initiate the exception handling flow. An example for an exception handler will be presented in chapter 9.

Using Variables as Parameters

Parameters do not only accept concrete arguments, but also variables. These variables represent values that are filled in during the execution of an action. This way, context-dependent information can be taken into account such as the currently failed peer service. Actions like “discoverService” can take this information to search for the service that has actually failed. Variables are indicated by a dollar symbol (“$”). For instance, to describe the name of a service, you can write “$serviceID”. The action receives information including the name of the lost service and internally replaces the variable by that concrete name. At the moment, DEEVOLVE only supports the replacement of variables indicating the service name.

8.4.3 Adaptive vs. End-User Exception Handling

The flexible phase model of DEEVOLVE (Figure 8-1) makes it possible to define adaptation methods that can be executed automatically upon the incidence of an exception. This feature facilitates peer operators to initiate and to run DEEVOLVE as a purely *adaptive* architecture. Adaptive in this case means that the system (i.e. DEEVOLVE) is solely responsible to perform the detection and the handling of exception stand-alone.

Relying on a purely adaptive software architecture is, however, critical. Depending on the working context, different handlers might turn out to be suitable to handle the exception. Often, these handlers can not be anticipated in advance. To tackle this problem, DeEvolve allows to define a set of pre-defined exception handlers at design time. Upon the occurrence of an exception, a user is able to select the most appropriate handler at run time. Moreover, if no handlers can be determined at design time, users are yet able to tailor a service composition to resolve the exception.

This thesis promotes the mixed variant to create exception handlers. Handlers can include automatic options, for instance, to pursue some initial actions for exception handling (e.g. notifying users). Whenever an option cannot be derived or described clearly, the component assembler should provide a range of options for the user.

8.4.4 Deployment and Execution of Exception Handlers

During deployment of a PeerCAT-based service composition, the DEEVOLVE environment of the service consumer also parses the internal parts that declare the excep-

tion handlers. The XML-based tree is then transformed into concrete Java objects. A handler is mapped by a self-contained Java object and stored internally in DEEVOLVE's handler service (Figure 6-18). The handler object holds further references to objects representing the declared options as well as the associated actions (see Figure 8-9 for understanding the structure).

A handler is addressed through the DEEVOLVE handler service API. This API is invoked by DEEVOLVE'S integrity checker service (section 8.2.5). Recall that this service is invoked after the detection of an exception, that is, the unavailability of a peer. If at least one integrity constraint of a currently active service composition instance is violated, a request will be sent to the handler service to invoke the declared handler for the given integrity constraint and for the given composition. Both the identifiers of an integrity constraint and that of the service composition are passed as arguments to the handler service. Based on these attributes, the handler service selects the internally stored handler object and extracts all option objects (see Figure 8-13). These objects are presented to the user within a Java Swing-based GUI dialog (see snapshot of it in section 9.5.2, Figure 9-13). This dialog also entails some information concerning the reason for the failure, an information which integrity constraints have been violated and so on. If a handler features autonomous options, status information for this option is provided and updated in the upper half of the dialog. From the set of available options, the user can now select the handler he believes to be the most accurate one.

Behavioral Aspects of Actions

Each executable handler action must implement the Java interface org.deevolve.integrity.handling.IAction to become a valid action. This interface provides various methods a concrete action has to implement, such as getter and setter methods for initializing the attributes that have been declared within an option. Thus, DEEVOLVE instantiates an option as an instance of a concrete action it associates together with the additionally defined parameters. So, if two handlers possess options referring to the same action, then two different action objects will be instantiated for each handler. The most important method of interface "IAction" is "compensate". This method is called by the DEEVOLVE handler service to invoke the option. This method is invoked together with a couple of further parameters that are handed over by the integrity service (including the name and id of the lost peer service, id of the violated integrity constraint). If the component assembler has used variables in the option declaration, the corresponding attributes will be taken instead (for instance to discover the lost peer service by its service name, see section 8.4.2).

While the handling procedure is running, the action implementation can use callback methods of the handler service to announce the progress of the handling procedure (see Figure 8-13). This is useful especially during long-lasting actions (e.g. during retrieval of alternative services). This feedback information is used by the handler service to refresh the progress bar in the handler dialog (Figure 9-13). The final result of the handling procedure is passed by invoking method "actionHasFinished". The result of an action is encapsulated in an object of class "ActionResult". If the option is part of an option sequence, then this object is passed to the subsequent option. The next option can make use of the previous results, if applicable. Whenever an action cannot be finished, the action invokes callback method "actionIsCanceled". In Figure 8-13, this method is called by the second action "DiscoverAction" after an unsuccessful attempt to locate an alternative service.

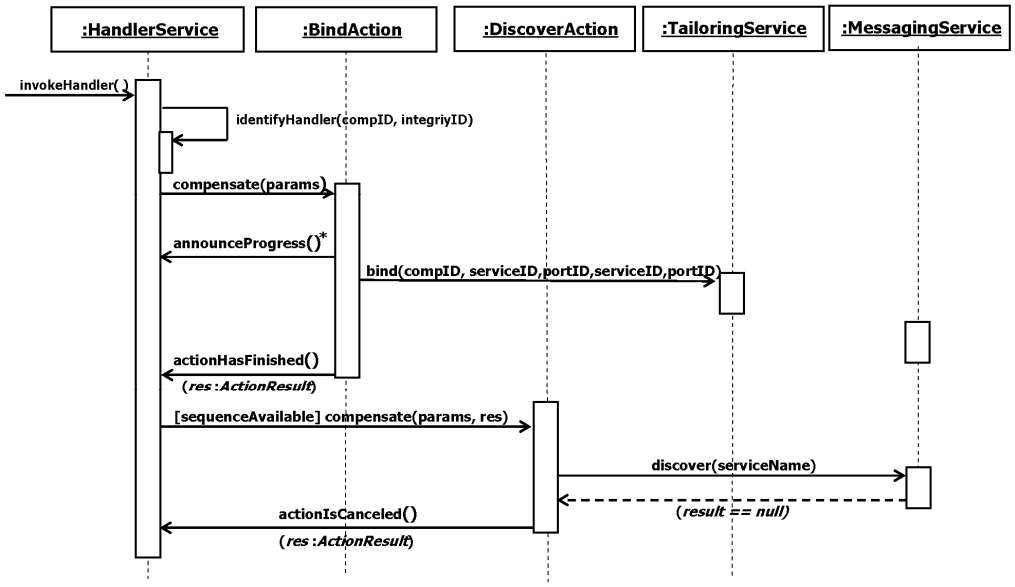


Figure 8-13: UML sequence diagram to visualize the interaction between DEEVOLVE's handler service and two options

Implementation of Adaptation Methods

Both actions from Figure 8-13 make use of two services provided by DEEVOLVE that provide concrete code implementing the adaptation methods. Action "BindAction" makes use of the tailoring methods being implemented in DEEVOLVE'S tailoring service (see section 6.6.1). This service merely conforms to the implementation of the Tailoring API of FREEVOLVE (see Table 8.1 in [Stiemerling, 2000]). The API has been extended in order to accommodate methods for tailoring service compositions (e.g. set a binding between two API ports, adding or deleting services from a composition). Upon receiving a request for adapting a service composition, DEEVOLVE's tailoring service identifies the internal "PeerServiceComposition" object that realizes the structure of the given composition (see Figure 6-12). It does so by calling the peer environment (parameter "compID" is passed by the action indicating the composition):

```
DeEvolveStore.getPeerServiceComposition( compID );
```

The obtained "PeerServiceComposition" object provides methods for adapting the structure of a service composition in a persistent way. The signature of these methods conforms to the adaptation methods as outlined in Table 8-2. After tailoring the structure, the adaptation methods are delegated to all registered "CompositionProxy" objects. These proxies eventually adapt the instances of that service composition at runtime. This way, any change to a running application can be seen directly on the screen.

Figure 8-14 demonstrates the delegation of an adaptation method from a "bind" action via the tailoring service towards the running component instances. The Proxy holds references to the concrete FLEXIBEAN component that allows it to manipulate them directly. In the example of Figure 8-14, a binding between an event producer (port of instance 2) is bound to an event sink (instance 1).

Action can also make use of Stiemerling's tailoring that adapt a peer service on its component level. Class "PeerServiceComposition" implements all these methods. When invoking these methods, an object of that class delegates the call to the corresponding "Component" object that represents the plan of the interface or the local composition (cf. Figure 6-8). This object then delegates the adaptation methods to all

its registered proxies, which then affect the changes to the FLEXIBEAN instances accordingly (see the sequence diagram in [Stiemerling, 2000], section 8.5.2, Figure 8.8 for a good illustration of the interplay among the involved objects).

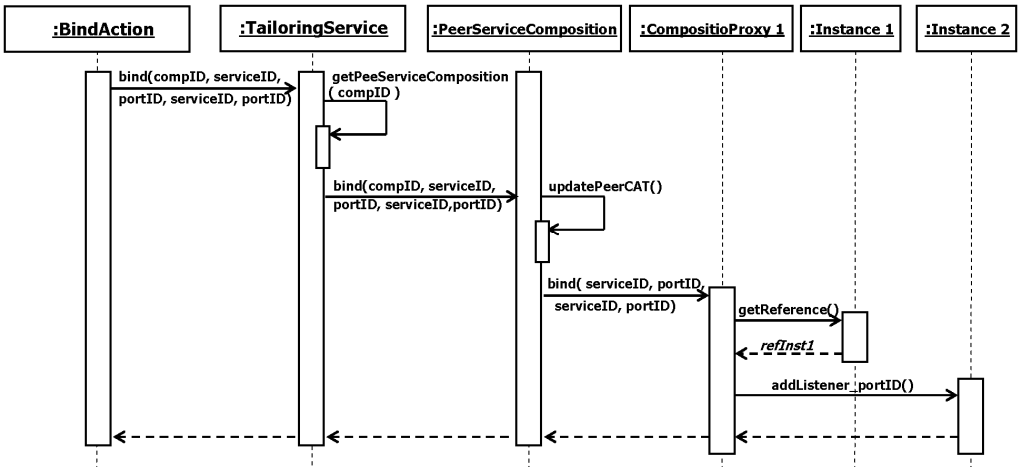


Figure 8-14: Sequence diagram showing the interactions among an action, the tailoring service and the composition objects

Action "DiscoverAction" makes use of DEEVOLVE's messaging service. This service encapsulates classes of the JXTA framework to send queries for finding advertisements describing peer services. The same messaging service is used by action "Notify" in order to send notification messages to registered peers.

Aborting the Execution of an Action

An action can internally throw an exception (org.deevolve.integrity.handling.ActionException) back to DEEVOLVE'S handler service to announce the unsuccesful attempt to execute an action or to signify any unnormal behavior within it. An example for an unnnormal behavior for action "discoverService" is the case when no peer service has been found in the peer-to-peer architecture. An error during the process of binding two ports (action "bindService") is another example for such exceptional behavior. Either case, the handler dialog (see Figure 9-13) is displayed again, so that the user again can select an appropriate new (or the same) handler.

Extensibility Aspects of the Action Implementation

In analogy to the integrity constraint approach, there is no upper limit concerning the number of actions that can be integrated and used in PeerCAT. New actions are plugged in dynamically as objects with respect to the strategy pattern.

8.5 Handling Exceptions at various Levels of Complexity

It has been a crucial requirement in this work to offer DEEVOLVE's adaptation method at various levels of complexity. This requirement fulfills the prominent demand of Morch to provide tailoring routines at different levels of complexity so that different user (groups) with varying skills can apply to those routines ([Morch, 1997], see also section 2.1.2.2 for a brief overview). In this work, two different phases have to make use of varying tailoring routines, that is, during the instrumentation of a service composition with exception handlers (design time) and during the actual handling (resolution) of an occurred exception during use time. As pointed out in section 8.1, different

actors come in the play during these two phases. A component assembler mainly plays an active role during the phase of instrumenting a service composition. During exception handling, an end-user (user) should play an important role as he becomes involved during the handling procedure. During both phases, power users are capable of refining the instrumentation or of performing advanced tailoring routines.

Level of Complexity (see [Morch, 1997])	**Adaptation Method (Actions) for the Instrumentation of PeerCAT**	**Actor**
Alternative Selection	All methods to involve user or consumer (invocation of tools for service discovery, tailoring tools, notifying external consumers), so that the user himself is asked to resolve the exception	Component Assembler
Construction of new Behavior	Defining (sequences of) options based on the adaptation methods for directly manipulating a service composition during use time (e.g. for discovering new services, binding services, adding or deleting services)	Component Assembler
Re-Implementation	Refining existing exception handlers with private, new handlers	Power User
	Preparing peer services with special code for internally resolving an exception. Ensures the exact and correct course of action to resolve an exception without the intervention of any user	Developer (of peer service)

Table 8-3: Overview on the three different levels of complexity to instrument a PeerCAT-based service composition with exception handlers

Morch proposes three different levels of complexity: alternative selection, construction of new behavior on the base of existing ones, and re-implementation. Apparently, the complexity increases from the first to the last level. During the instrumentation of a PeerCAT-based service composition, the provided adaptation methods serving as actions to resolve an exception can be assigned to one of these three levels as depicted in Table 8-3. The adaptation methods for actively involving a local user are interpreted as alternative selection, because the component assembler (mostly) only has to reason about choosing exactly one of these alternatives. However, he is disburdened to create a useful combination of these methods. Moreover, the complexity and the intuition to resolve an exception are shifted from design time to the use time (i.e. to the user).

Defining (sequences) of options based on the adaptation methods for directly manipulating a service composition at use time requires more intuition and more experience, which makes it a complex task. Here, the context in which these operations will be performed, must be anticipated in advance. For both levels, the component assembler is the actor carrying out the instrumentation.

For the last level (re-implementation), a power user is able to refine and, hence, to replace existing exception handlers by new ones. The ability of developers of a single peer service to implement code for internally resolving an exception is also placed on this level of complexity. Here, peer services could make use of the occurrence of an exception. Given an exception, the broker or the integrity checker service could also delegate (e.g. through a dedicated callback method) an exception object to the affected interface part of a peer service. This exception objects could contain all necessary information about the occurred exception or the violated integrity constraint. This option

would accomplish an exact and correct course of action to resolve an exception without the intervention of a user. Although practical, this work did not focus on the realization of this callback mechanism. Of course, a developer is also free to implement code for handling any kind of internal exception that might occur within a peer service (e.g. null pointer exceptions).

Level of Complexity (see [Morch, 1997])	**Adaptation Method (Actions) for handling an exception at runtime**	**Actor**
Alternative Selection	Selecting a predefined (sequence of) option(s) to modify a service composition directly.	User
Construction of new Behavior	Selecting an option that opens a local tool, e.g., for tailoring a service composition or discovering a new service. The user is then responsible to tailor the composition at runtime in order to resolve the exception	Power User
Re-Implementation	Re-implementing the services of a service composition	Power User / Developer

Table 8-4: Overview on the three different levels of complexity to handle an exception in a PeerCAT-based service composition at runtime

Table 8-4 shows the allocation of adaptation methods used to handle an exception at runtime. At first, users are able to select upon pre-defined options that refer to any kind of adaptation method for directly modifying a service composition. Although the execution of an option itself is complex, it is quite easy for a user to select an option. Choosing an option featuring an action to open a local tailoring tool, however, is clearly more complex, because a user is more involved in the resolution process than in the options from the previous level. A power user is the dedicated actor to carry out these adaptation routines on the level of constructing new behavior.

Note the different levels of complexity in the instrumentation phase and in the exception handling phase. For the component assembler, preparing an option is more complex than just selecting the same one by a user. Vice versa, adapting a service composition with a local tool is far more difficult for a power user than just declaring the usage of a tool during the instrumentation phase. Again, the last level enables more sophisticated power users (or the developer) to re-implement service being part of service composition. This is certainly the most complex alternative. As DEEVOLVE strives for an open-source realization, all source code of services are available, making this option definitely practicable.

8.6 Related Work

The related work that compares the approach of this section is divided into three parts: generic models for exception handling on an architectural level, component-based runtime environments, as well as aspect-oriented approaches for exception handling.

Generic Models on Exception Handling in Software Architectures

Oreizy and colleagues describe an architecture-based approach to self-adaptive software [Oreizy *et al.*, 1999]. This paper is a highly cited paper mentioned in many subsequent contributions on exception handing in software architectures. As one of the first papers, the authors outline general properties and demands on self-adaptive soft-

ware, which are motivated by a concrete scenario (flight mission control). Based on these assumptions, they propose an infrastructure that deploys component-based adaptation methods that are affected on an architectural level (i.e. manipulating components, connectors). They assume a dynamic distributed architecture based on the architectural styles C2 and Weaves (see section 2.5.3) that facilitates the runtime manipulation of components and connectors. Each physical site in a distributed architecture deploys an *architecture evolution manager* (AEM) that maintains the consistency between an adapted architectural model (i.e. a declarative description) and its corresponding executing implementation. This manager also features *change transactions* that can be composed of two or more basic operations. All changes are atomic, that is, they are either complete without error or leave the running application untouched. Furthermore, an AEM can deploy constraints that entail, for instance, the availability of a distinct component. Adaptation management embraces all activities for collecting observations from the environment, planning changes, and deploying the change descriptions to local sites (i.e. to the local AEMs). All these activities are mastered and carried out by agents, turning their infrastructure to an agent-based system. For instance, the *expectation agent* is responsible for collecting (complex) events from its environment (e.g. from an application, availability of network blocks) that could serve as an indication for *planning* adaptations. Adaptation are described in terms of change descriptions that are propagated by so-called *change agents* to each affected physical site. Here, these agents interact with the local AEM to finally pursue the adaptations.

Apparently, Oreizy's approach is very complex and general-purpose and can, therefore, be adopted to many architectural styles and application scenarios. In fact, some concepts of DEEVOLVE are similar to his approach. DEEVOLVE ensures the above aspired consistency between the architectural model and its running implementation by DEEVOLVE's component and proxy approach: each adaptation to a component (representing the template) is directly propagated to its registered proxies that in turn affect the running Java objects. What is clearly missing is a transaction model for keeping the consistency when pursuing aggregated adaptation (see also additional remarks in section 4.4). A proxy object can be compared to change agents with respect to Oreizy. The AEM in his approach features an integrity constraint as conceptualized for DEEVOLVE. However, it remains unclear, if the approach deploys single fixed constraints, or if further new ones can be added as in DEEVOLVE. The obvious benefit of DEEVOLVE is that a local integrity constraint can also incorporate the states of external elements of an architecture, that is, external peer services. Although Oreizy strives for user involvement in his system (e.g. for providing further input to expectation events), it seems rather unclear when exactly user involvement can occur in this model. Obviously, there is no designated user involvement prior or during the planning or execution of adaptation changes. Involving users during these phases is a clear surplus value of DEEVOLVE compared to his approach. Despite the general-purpose character of Oreizy's architecture, it seems that his approach is rather inappropriate for the adoption to application scenarios with a high degree of user presence and uncertain contexts, in which exceptions can occur.

Dellarocas also proposes a rather general-purpose model for handling exceptions on an architectural level [Dellarocas, 1998]. He suggests that the whole system, as well as each participating component should have a model that specifies the correct *behavior* of both the system and the components. Another model must be available for specifying incorrect system behavior. A further component type, so-called *sentinel components,* guards the interaction between components. Based on a classification approach, a sentinel component is capable of detecting exceptions by evaluating both types of

models mentioned before. Upon detection of an exception, a handling process is invoked that incorporates an exception resolution phase (see Figure 1 in his paper). During this phase, a user can be involved. Adaptation methods are merely based on behavioral aspects (e.g. undo or redo an operation, change of resource requirements).

The phase model of Dellarocas is comparable with the phase model of DEEVOLVE (see Figure 8-1). It also features phases for preparing a given piece of software during design time, while phases for detecting and resolving exceptions are also carried out at runtime. A major difference to his work is that DEEVOLVE's adaptation approach totally relies on structural composition, whereas the internal behavior is not regarded. In Dellarocas' approach, the complete behavior of a component-based application must be anticipated in advance, which could be a non-trivial if not impossible task, especially in application scenarios with complex contexts. During user involvement, the user is supposed to possess adequate knowledge on system behavior. Presumably, he could be overstrained with the demand to select appropriate handlers during the resolution phase.

Based on available general-purpose models for exception handling (i.e., mostly based on the work of Oreizy), numerous approaches of handling exceptions on an architectural level have been proposed (cf. [Ben-Shaul *et al.*, 2000] [Cheng *et al.*, 2002] [Dashofy *et al.*, 2002] [Badr *et al.*, 2002] [Bialek and Jul, 2004]). The common goal of all these research projects has been to suggest completely adaptive solutions without any user involvement. Obviously, some commonalities do exist between their approaches and DEEVOLVE's model for exception handling. They are omitted here for the sake of brevity.

Component-based Runtime Environments

A large number of research projects on component-based runtime environment have been proposed for different architectural styles, constraints, and application scenarios. The task of presenting and analyzing a complete overview of state of the art engines would certainly go off the scope of this thesis. Two approaches, Gravity [Cervantes and Hall, 2004] and P2PCom [Ferscha *et al.*, 2004], are worthwhile considering since they combine both the component-oriented and service-oriented model to a coherent model suitable for distributed architectures. The diploma thesis of Palij provides an extensive comparison of these two approaches and the DEEVOLVE platform [Palij, 2006]. In the following, the core results of this analysis are outlined briefly.

The foundation of Gravity is a service-oriented component model, in which components implement the contract of a service. Furthermore, the architecture supports the declarative composition of different services, as well as the dynamic substitution of unavailable services with other compatible services. Services can be indicated as optional, denoting that the loss of such a service does not invalidate the overall composition. The latter concept is similar to the minimal composition integrity. In contrast to Gravity, DEEVOLVE provides a more flexible approach, as new integrity constraints can be added to the runtime environment dynamically. Gravity assumes a centralized architecture, whereas an adoption of the concepts towards a distributed architecture is planned and certainly realizable. Although exception handling has been implemented (reaction to a lost service), the handling capabilities of Gravity are rather limited. Basically, no options can be defined within a handler that are to be executed autonomously or selected manually by a user. User involvement is not regarded as a research issue. However, due to initial similar concepts and due to the sound foundation and

implementation of the concepts (it relies on the OSGi platform), Gravity constitutes the most related project seen from a technical level.

The P2PCom architecture combines recent aspects of peer-to-peer architectures (where peers can also represent small, resource-constrained appliances such as a PDA) and the component-based methodology for reusing existing code (implemented and provided by components). One of the major goals has been to dynamically discover, to inspect, and to compose components to new applications in the presence of component failure. From a technical point of view, this concept has been realized by JXTA as the appropriate peer-to-peer framework and by OSGi as the component framework. Since OSGi containers only support the interaction between local components within one container (remote interaction must actually be mastered by the components themselves), P2PCom introduces a novel port concept for embracing remote capabilities. Here, a component can communicate with remote components by using a *port manager* that mediates a port call between components of different containers (section 3.3 and 3.5 in [Ferscha *et al.*, 2004]). The communication between different containers is mastered along so-called *access ports*. These ports are implemented by using the pipe technology of JXTA (see section 2.2.5). The port manager is responsible for detecting peer failures and, given an occurred exception, to detect and bind in an equivalent service (offered by a component) based on service contract specifications. This technique is indicated as *hot-swapping*.

Although P2PCom is targeted for a peer-to-peer architecture, it provides no aspects for involving the interests of a user. The proposed concept of the handling mechanism including the hot-swapping technique conforms to a purely adaptive system. The unavailability of components is the only type of exception being supported, a concept for detecting exceptions on a semantic level (in DEEVOLVE supported by integrity constraints) is not realized, either.

In the beginning of this research project, two further ambitious open source projects have addressed the combination of both service- and component-orientation, JINI [Arnold *et al.*, 1999] and Avalon[24]. Although the concepts and implementations have been well founded, both projects have no more relevance in the state of the art.

Aspect-oriented Approaches for Exception Handling on an Architectural Level

In the (relatively) new and popular research field of *aspect-oriented* development [Kiczales *et al.*, 1997], code for handling exceptions on an architectural level is often interpreted as a *cross-cutting concern*, since it touches many independent components of the overall system. Aspect-oriented systems provide means to *weave* such a cross-cutting concern into the component (realizing the *core concerns*) that depend on the functionality of that cross-cutting concern. The application of aspect-oriented methods to realize exception handling in a service-oriented architectures that is suitable for ambient application scenarios is described in ([Rho *et al.*, 2006]). This approach has certainly many commonalities to the approach of DEEVOLVE, although the fundamental approach is different. A sound model for user involvement at distinct decision points is (yet) missing in this project. Owing to the huge success of aspect-orientation, future tailorable and adaptive architectures will clearly fall back on these technologies.

[24] see the last (and *sad*) web page of Avalon: http://avalon.apache.org/closed.html

8.7 Summary

This chapter has summarized the main aspects of DEEVOLVE for handling exceptions in a service-oriented peer-to-peer architecture. The fundamental exception type DEEVOLVE is capable of handling is the unavailability of a dependent peer. This type of exception might lead to the violation of integrity constraints that declare additional conditions on a service composition, or on remote peers (consumer or provider). Mechanisms for handling exceptions correspond to actions that are based on component-based adaptation methods. Most adaptation methods conform to the methods as proposed by Stiemerling's work. On top of these, this work has suggested additional methods that are of particular use in the context of service-oriented architecture (discovery of services, subscription to services, deployment of services). Furthermore, the instrumentation of a service composition with exception handlers as well as the handling of an occurred exception are provided on different levels of complexity. End users with different technical background skills are now able to adopt these methods.

Chapter 9
Evaluation of DEEVOLVE in the COBE project

This section aims at presenting the results of the CoBE project, in which fundamental aspects of this dissertation project are evaluated. The evaluation consists of presenting application scenarios stemming from the area of structural design. The goal is to show, how the DEEVOLVE architecture (i.e., especially the exception handling mechanisms and the notion of integrity constraints from section 8) is able to enhance and to improve the effectiveness of such scenarios. At first, typical characteristics of today's projects in construction engineering are outlined in section 8.1. These assumptions have lead to the initiation of the CoBE project, whose goals are summarized in section 8.2. The fundamental contributions of the CoBE project are outlined in section 8.3. Finally, three different application scenarios are presented in 8.4. Based on these scenarios, the capabilities and the strengths of the DEEVOLVE architecture for supporting processes in construction engineering are elaborated. The section closes with a conclusion on the COBE project.

9.1 Characteristics of modern Projects in Construction Engineering

Large-scaled projects in the area of construction engineering are nowadays organized in a decentral or virtual manner, typically indicated as *virtual organizations* (VOs) [Barnatt, 1995]. Virtual organizations assume the involvement and close collaboration of different *partners or experts* (e.g. engineers, technicians, or draftsmen), *facilities* (e.g. local authorities, customer's offices, construction sites), and further *computational resources* (e.g. hardware for executing finite element (FE) simulations, expensive devices such as document plotters). The participants typically reside at different geographical locations. Although each participant of such a virtual organization may define individual sub-goals (e.g. the preparation of a partial structural model, the verification of a finite element model, the observance of technical guidelines), the entire organization works towards the achievement of a *common goal* (e.g. final consistent structural model of a steel bridge). Despite the independence of the constituting partners of a virtual organization, there is no contention among them [Kämpf, 2005].

The adoption of virtual organizations as the default structure of large engineering projects can be justified by prevailing circumstances that can be figured out in this engineering discipline, that is, complexity, competition pressure, high quality demands and the reduction of expenses [Alda, 2005b]. Virtual organizations accomplish the reduction of expenses by enabling individual partners to participate in many different projects simultaneously without leaving their office. Owing to the lack of a corporate

administration or head office, expensive costs for physical environments like buildings, meeting rooms can be saved.

The success of virtual organizations depends on various factors or requirements. Rittenbruch and colleagues identify three different requirements that are crucial for a virtual organization [Rittenbruch *et al.*, 1998]:

- Trust among the individual partners of a virtual organization
- Flexible structures that allow to react to new demands and situations
- Communication, Coordination, and Cooperation among the partners

Trust as a value expressing the reliability of partners[25] is justifiably significant for projects in the area of construction engineering. Especially in large project settings, different engineering projects need to collaborate that may have not worked together before. In addition, the involvement of new partners in existing structures must be based on a trust level to ensure the reliability and competency of those new partners.

The request to have flexible organizational structures must be taken into consideration without fail. Virtual organizations within construction engineering usually exhibit a *dynamic structure*. Project partners join and leave the organization depending on their dedicated role in the virtual organization or due to project-related circumstances. For instance, a structural engineer usually comes into play during the initial phases of construction projects in order to verify and to detail structural elements. The same engineer could later rejoin an organization after the occurrence of a disaster (in German: *havarie*) in order to re-validate the statics of important building panels (see more subtle havarie scenarios in [Holz *et al.*, 2002]). Partners could also leave an existing project work due to further individual reasons, e.g. insolvency. Besides these rather medium-term reasons, there are also short-term or technical reasons for leaving a co-operation, for instance, evoked by temporary network connections. Owing to the dynamic availability of partners and their resources, effective planning activities often cannot be ensured in such virtual project structures. Another issue one has to regard in this context is the generation of closed sub groups within the overall project constellation. These sub groups may result if various partners are willing to carry out important design tasks synchronously without any interruption from other partners.

The establishment of Rittenbruch's third requirement, that is, communication, collaboration, and coordination as a mean for regulating the work among all involved partners is certainly the most challenging one. The achievement of these three aspects necessitates the organization-wide introduction of appropriate software tools and infrastructures [Kämpf, 2005]. During the last years, modern group-supporting software systems such as groupware systems (see [Schwabe *et al.*, 2001]) for an overview), traditional email clients (e.g. MS Outlook, Modzilla, Thunderbird), document configuration and exchange tools (e.g. CVS, Email), shared workspaces (e.g. BSCW [Fraunhofer, 2005]) or chat systems (e.g. ICQ [ICQ, 2005]) have been utilized for supporting these co-operations. These systems enable partners to share CAD documents (e.g. through Mail or BSCW), to negotiate important design decisions synchronously (e.g. chat tools, video conference) or to derive an accumulated view of the project progress (e.g. agenda systems, whiteboards).

One of the major observances the CoBE project has made is that these state-of-the-art off-the-shelf systems are not sufficient for supporting networked co-operations in

[25] Besides trust, reputation is another important value to assess the general opinion of the public towards a person or a group of people (see section 2.2.3 for an overview)

construction engineering [Alda *et al.*, 2006]. Most systems do not consider the *heterogeneous* technical environments of the dedicated partners. What one has to expect is that engineers insist on working with proprietary *engineering software* (e.g. AutoCAD200x, VarioCAD, BoCAD), use incompatible *document exchange formats* for product models (IFC 2.x, CIS/2), and various hardware infrastructures and different *access abilities* to the Internet (e.g. Modem, DSL). Mostly, partners customize their own working practices, that is, they work *asynchronously* (i.e. work time-shifted or from different locations), or hide working details due to of *privacy* concerns. Most systems are also not suitable to cover the typical dynamic structures of virtual organizations, e.g. to handle the unavailability of partners during synchronous activities.

Although the premises of virtual organizations have been recognized and fairly understood, the above-mentioned problems yet constrict the fully appreciation of them in construction engineering. What are clearly needed are *holistic* solutions for supporting such project constellations that hold special demanding requirements as reported in this section. The CoBE research project as part of a priority research program has taken these problems of virtual project constellation in construction engineering as the initial point for finding new ways for supporting those projects. The goals of CoBE are outlined in the next section.

9.2 Goals of the CoBE project

The CoBE (***Co***mponent-Based adaptable Platform for networked cooperations in Civil and ***B***uilding ***E***ngineering) project has been initialized as part of the priority research program PP 1103 "Network-ba sed co-operative planning processes in structural engineering" funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) in 2001. The aspired overall goal of this priority program has been to

> *"...re-design the planning processes of structural engineering for the utilization of distributed resources, to develop adequate co-operation models for the technical planning by use of information sharing between project partners and to allow co-operative project work with distributed technical models in networks*." ([Rüppel, 1999], English abstract)

Planning projects in construction that employ these goals are referred as *networked co-operations*. Networked co-operations emphasize the *network-like* collaboration of different dispersed partners in planning projects. In fact, networked co-operations correspond to and hold similar characteristics as virtual organizations (see previous section). In contrast to virtual organizations, networked co-operations aim at focusing on the technical integration of *specific* software and resources of partners.

In the beginning of this research program, three goals were identified in the terms of a research or working schedule (see again [Rüppel, 1999] for a complete overview):

- Specification of *software platforms* supporting the communication among participating stakeholders within a networked co-operation including new ways for the *integration* of *specific engineering software* and models into the resulting network
- Development of *process models* for networked planning processes that takes into consideration the respective properties of networked co-operations
- Proposal and development of standardized *product models* for the problem-oriented usage of information within a networked co-operation

The COBE project has mainly committed for the first research item, that is, the specification of a software architecture supporting the *communication* among participants in a networked co-operation. An essential requirement of that platform has been to reason about and to integrate new approaches accomplishing the flexible *adaptation* of deployed software in such architecture. These adaptation mechanisms should facilitate to accommodate new requirements of software or to react to changes in the project environment (e.g. the failure of a dependent partner or remote software). During the first phase of the project, additional goals have been defined that contribute also for the *coordination* of members within a networked co-operation.

A first goal of the project has been to elicit concrete *functional* and necessary *non-functional requirements* for both a software platform and a coordination model supporting networked co-operations in construction engineering. In order to gain a better understanding of the typical characteristics of planning scenarios in construction design, a concrete visionary scenario based on a real-world structural design project has been developed. Based on that scenario, fundamental requirements have been derived. These requirements serve as an input for selecting an appropriate architectural style. This scenario is presented in the next section 9.3.

9.3 Result of the Requirements Analysis

This section presents a *visionary scenario* that serves as a base for understanding the realities of a project in structural design and for analyzing principle requirements for a software architecture. Figure 9-1 depicts a visualization of that scenario.

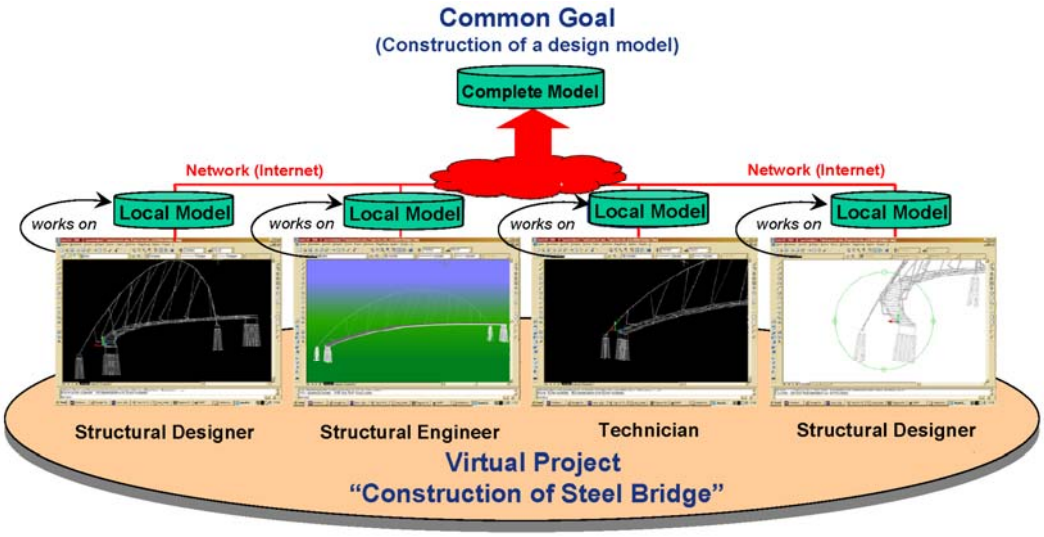


Figure 9-1: Principle project constellation for a networked co-operation

In the depicted scenario of Figure 9-1, four different members have joined a co-operation to develop a structural model of a steel bridge. The achievement of that design model constitutes the common goal of that co-operation. Each member is equipped with a CAD program, with which each member is capable of modeling a local, that is, *partial model*. The fusion of all partial models corresponds to the complete (total) model. This way, the partial models are coherent, that is, they depend on each other. The consistency of all partial models at given points in time (e.g. before a milestone arrives) is a crucial criteria within a co-operation. Consequently, any member of a co-operation does not model in isolation but in relationship to all other mem-

bers. The unavailability of a single member and (thus) the member's models hinder all other members to validate[26] their new modeling steeps with the remaining depending models. This is in particular a problem for synchronous collaboration, where members carry out their design activities at the same time *in parallel*. In this case, the consistency of all partial models cannot be guaranteed entirely.

On the base of the illustrated scenario, a *use case model* could have been derived that presents an abstract presentation of the functional as well as non-functional requirements of a suitable software platform supporting dispersed planning activities. This use case model does abstract from the actual circumstances of a concrete scenario and presents a more general view on the aspired requirements. This resulting model has already been demonstrated in terms of both as a graphical UML use case diagram (Figure 2-5) and in a textual manner in section 2.4.1. That section has followed a circumstantial discussion concerning the software architectural style being appropriate for a software platform supporting above-illustrated application scenarios.

The $SO_{P2P}A$ architectural style as well as the DEEVOLVE platform constitute the major achievement from that project. Seen from a scientific perspective, the focus of this work has been to establish new approaches for realizing the essential non-functional requirements *adaptability* and *reliability* for such an architecture. The results have already been presented at length during the previous chapters. The purpose of the next section 9.4 is to rephrase the contributions of this work in order to ascertain, how they contribute to the aspired goals of the CoBE project.

9.4 Results from the CoBE project

In this section, the main contributions of the CoBE project are presented. The software architecture DEEVOLVE thereby serves as aspired software platform for supporting networked co-operations in construction engineering (section 9.3.1). On top of this architecture, the COBE AWARENESS FRAMEWORK has been developed, which realizes a decentralized coordination model based on the sole awareness of planning activities (section 9.3.2). In the last sub-section 9.3.3, some further developments are illustrated, mainly considering the integration of DEEVOLVE with autonomous behavior (provided by software agents).

9.4.1 DEEVOLVE as the Solution for a Software platform

The DEEVOLVE architecture is proposed as the software platform supporting the communication and collaboration of members within a networked co-operation in construction engineering. Since DEEVOLVE has been presented at great length during the previous sections, only those aspects are highlighted that have not been covered so far and that are in the context of the special requirements of the COBE project.

9.4.1.1 Principle Aspects

According to the principle notion of DEEVOLVE, the organizational structure of a networked co-operation is mapped to a software architecture that consists of many independent peers. Each peer represents a single member or facility within such a co-operation. From a technical perspective, each co-operation member is equipped with a

[26] For instance, by exchanging information on partial models or exchanging the complete model.

single local DEEVOLVE peer runtime environment. Under the assistance of the DEEVOLVE environment, the team member can locate, deploy and use various remote applications (peer services) provided by other members of the co-operation. As being a peer provider, he is also able to offer own applications (services) to other members.

DEEVOLVE supports four different ways with different complexity for enabling end-user to develop and to provide peer services.

- Regular members can directly use and publish one of the predefined peer services available in each DEEVOLVE environment (like the "*DocExchange*" peer service see section 9.4.3). Here, no adaptations are necessary.
- More skilled users can make slightly adaptations to a predefined peer service such as adapting the skin (look and feel) of it. Here, no major adaptations are undertaken, that is, the functionality remains unchanged.
- Sophisticated users are able to compose an existing or a new peer service by the composition of several provided peer services. Alternatively, a user is always capable of adding and deleting single peer services and single components. This way, the functionality of a peer service can be changed.
- Technical experts (e.g. construction informatics) are able to develop own, completely new services that must follow the FLEXIBEAN component and decomposition pattern. DEEVOLVE supports the creation and publication of advertisements in order to announce the availability of that service.

The underlying peer-to-peer idea allows members of a networked co-operation to work and to maintain their partial design models independent from any kind of central server. Thus, the co-operation remains independent from any central authority. By offering their individual design models into the co-operation, the constituting peers establish a *virtual shared workspace*. Being a "peer" within a co-operation, each member is able to have *controlled* access to this shared workspace. This feature allows for exchanging documents (e.g. a CAD file) or for sharing information on the status of certain modeling activities within a co-operation. A controlled access is reached by allowing peers to self-organize into self-governed peer groups (see section 9.4.1.3)

The proposed architectural style of DEEVOLVE emphasizes the *autonomy* of single members. An engineer is still capable of working on design models even if he has no connection to a network (e.g. on a construction site). The style also respects the potential *reluctance* of engineers to share certain design models and decisions with other partners or central servers.

9.4.1.2 Bridge Components to integrate Legacy Applications

DEEVOLVE imposes the FLEXIBEAN component model for the development of purely local as well as distributed applications, that is, applications consisting of a local and a remote part. For demonstration purposes, several tools have been implemented like the CAD Whiteboard application, which is based on the JHotDraw drawing framework[27]. For a more practical utilization, however, it is absolutely required to integrate existing mature standard CAD applications like AutoCAD (legacy applications). Owing to the Java-based foundation of the FLEXIBEAN component model and due to the C++/COM-based solution of AutoCAD, this goal appears unfeasible at first.

In the course of the project, so-called *bridge components* are used for mediating between components of incompatible component models. For the given requirements of

[27] See http://www.jhotdraw.org/ for more information

the CoBE project context, a special bridge component called COM2Java has been produced that mediates between a FLEXIBEAN component and a COM-based COTS[28] component. So, in order to be deployable in DEEVOLVE, an application must provide an open programmable interface that allows to access internal functionality of the application. Closed applications cannot be accessed from outside and, thus, cannot be deployed in DEEVOLVE. Since AutoCAD version 2002 provides such a COM-based interface, it has been possible to integrate this standard application into a DEEVOLVE peer environment (see Figure 9-2).

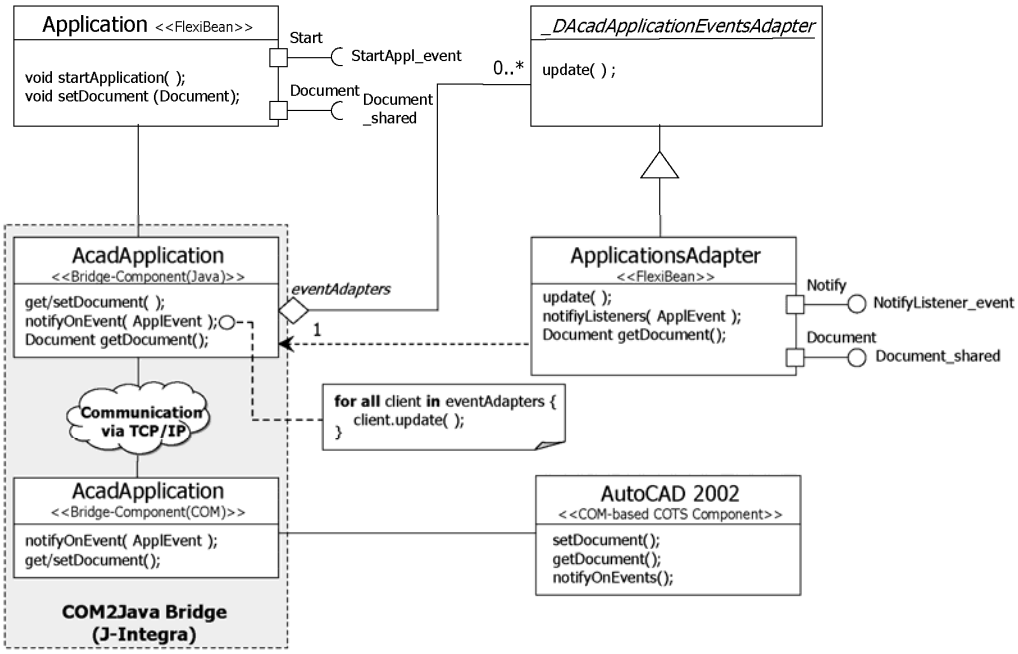


Figure 9-2: COM2Java bridge for mediating between FLEXIBEAN components and a COM-based COTS application (here: AutoCAD 2002)

The implementation of the COM2Java bridge is based on the commercial J-Integra framework [Intrinsyc, 2005] and on experiences from the COM2ACAD framework developed by Bilek and Katzmarzik [Katzmarzik, 2003]. The bridge component itself comprises two parts, a Java-based and a COM-based sub component. The communication between these two sub components is performed via the TCP/IP protocol (Figure 9-2). Thus, every call to a method to the FLEXIBEAN component Application is delegated via the bridge component to the actual AutoCAD 2002 component. Through this component, the AutoCAD application can be initiated and started with a given document (represented by class Document, a Java-based representation of the internal AutoCAD format). The Java interface of that bridge has been generated by the J-Integra compiler. This compiler takes the interface description file of a COM-based application and delivers a set of Java classes that represent this interface and that make it accessible from any Java client.

Another FLEXIBEAN component called "ApplicationsAdapter" uses the event notification mechanism of AutoCAD. This mechanism fires events whenever certain applications events have occurred (e.g. the closing of a drawing, the manipulation of a model). All these events are usually triggered by user actions. This way, any component or application connected to the "ApplicationsAdapter" can be notified about

[28] Commercial Off-The-Shelf = a ready made component

events from within a local AutoCAD installation. In addition, the currently active model can be obtained through the shared object of that component.

The above-illustrated component-based application can easily be enhanced to a publicly available peer service that can be located within a given already established peer-to-peer architecture. Once located and bound to the environment, this peer service allows for sharing arbitrary CAD models among members of a co-operation. Moreover, emitted events from within an AutoCAD environment can be delegated to bound peers. This way, any member can perceive information on modelling activities of other third-party members. Based on that information, he is then capable of deriving new and further own activities. This interaction paradigm serves as the foundation for CoBE's coordination model, the so-called COBE AWARENESS FRAMEWORK that will be explained in section 9.4.2.

9.4.1.3 Peer Groups for defining Boundaries of Co-operations

The collaboration of several peers leads to the establishment of networked co-operations. In theory, each peer (i.e. member) is able to interact with all other peers (members) according to the idea of the peer-to-peer paradigm (see Figure 9-3).

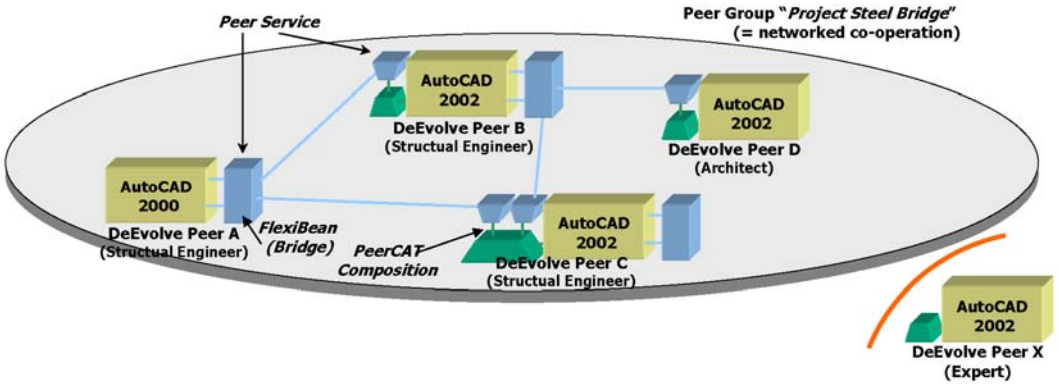


Figure 9-3: Scenario of a networked co-operation based on DEEVOLVE

This liberal model, however, raises questions concerning the scalability and the security of such networks. In DEEVOLVE, the notion of "peer group" is used to define the boundaries of a networked co-operation. Peer groups accomplish peers and, hence, their dedicated members to collaborate with each other protected from any unauthorized peer (peer X in Figure 9-3). A new peer willing to become a member in an existing group must first apply for membership. This application has to be accepted according to the predefined membership policy of that group. For a wise solution, however, the "AcknowledgeMembership" policy should be taken (see section 6.2.3). This policy allows the founder of a group to acknowledge the requesting peer.

In DEEVOLVE, any peer is able to generate groups. A peer may naturally belong to many groups. So, a member can belong to many groups and, consequently, to many networked co-operations and projects in parallel. Within a group, several sub groups can be initiated. The founder of a group in general takes over some responsibility within a construction project, as for instance, an architect or project manager.

9.4.1.4 Integrity Constraints for ensuring Consistency

The dynamic availability of partners often limits the effectiveness of planning activities in networked co-operations. From the viewpoint of a DEEVOLVE peer environment, unavailability of partners also means the unavailability of peer services and,

often, the untraceability of resources (e.g. a partial design model). In such a case, a single partner is not able to share resources and, hence, to collaborate with potentially important partners. The progress of individual planning activities could also be hindered by the absence of important value-added peer services such as a model checker service. In all cases, the consistency between dispersed planning activities and between partial design models cannot be ensured any more.

The operator of a DEEVOLVE environment is capable of defining integrity constraints on peer services and service compositions. Integrity constraints can be regarded as a *contract* between the different stakeholders of a networked co-operation. A contract defines the quality of service each partner has to adduce in terms of availability of his peer environment and, thus, of his provided peer services. If an integrity constraint has been formulated between a number of peers, then all affected peers can ensure the availability of services and resources within a dedicated working context (e.g. the construction of a single bridge element). If a given *integrity constraint* is fulfilled, then the *consistency* of the partial design models can be ensured as well. The integrity constraint is said to be violated, if at least one involved peer has become unavailable. For each integrity constraint, appropriate handlers can be associated that are executed in case of an integrity violation. Here, the operator of a peer can be involved during the phase of exception resolution (see section 8.4).

The declaration of an integrity constraint represents a *discrete state* a networked co-operation may enter during its lifetime. Although it is conventionally up to each peer to anticipate such states, the initiator of a co-operation (peer group) appears more appropriate to figure out important states and to define appropriate integrity constraints (and handlers). He is then able to define these conditions and to distribute them to corresponding peers. These peers can interweave these conditions in their local peer service composition (PeerCAT).

Each peer operator is able to express a reputation value for peers from which he has used peer services. Given the incident of exceptions or integrity violations, he can naturally express a negative reputation value and publish it as an advertisement to its project leader (i.e. a rendezvous peer). The project leaders then disseminate these advertisements across other rendezvous peers throughout the peer-to-peer architecture (see section 6.3.6 for more details). Reputation values are worthwhile to consider for a project leader during the startup of new co-operations. Based on the gained reputation advertisements, a peer is able to decide whether to trust a peer (operator) for future collaborations. Apparently, defining and publishing reputation values is not implemented in DEEVOLVE but suggested by the $SO_{P2P}A$ architectural style (section 4.1.2). It is future work for DEEVOLVE, which could clearly benefit the constellation of new project structures.

9.4.1.5 Adaptation Policy for guaranteeing homogeneous Resources

The design of a bridge component accomplishes the integration of various legacy applications. Given the availability of an appropriate bridge, each application written in an arbitrary programming language can be included and published as a peer service. So, the DEEVOLVE platform can guarantee to have a homogeneous co-operation in which all local heterogeneous software installations can be accessed. The operator of a peer, however, still possesses the freedom to update local installations or to adapt peer services according to private requirements (e.g. by deleting certain functionality). While such adaptations to a local environment may necessarily improve the efficiency

at a single partner's site, it may violate context dependencies to other team members, whose local peer environments depend on the original service version.

The founder of a peer group, that is, the operator of a networked co-operation is responsible to define *adaptation policies* (section 7.3) in order to regulate adaptation requests within a given networked co-operation (i.e. peer group). Adaptation policies can be defined according to the current project status. Towards the end of a (critical) project, the architect could update the adaptation policy imposing a more restrictive dealing with adaptation requests (e.g. no adaptation is allowed until the project's end).

The disrespect of an imposed adaptation policy by provider peers can potentially reduce their reputation within a peer-to-peer architecture (i.e. within a given established engineer community). Given such case of disregarding a policy, affected consumer peers are able to argue a negative reputation value against this provider. Likewise to reputation statements when violating an integrity constraint, these statements can limit the trust of other peer operators to work with such designers in future projects.

9.4.2 The COBE AWARENESS FRAMEWORK

The theory of awareness for regulating distributed activities has become a popular research topic in the area of Computer Supported Cooperative Work (CSCW). According to the author Dourish, awareness is an understanding of the activities of others, which provides a context for your own activity [Dourish and Bellotti, 1992]. With the introduction of awareness in groupware systems, each user is equipped with mechanisms, with which he can be aware of activities of other users belonging to a common activity or a common data set. For example, if a user is modifying a shared document, a predefined number of users will be informed about all subsequent changes. For receiving notification events about status changes, a user first has to subscribe to a notification list. This kind of awareness is called *task-oriented awareness*. Awareness can be regarded as a foundation for conflict recognition and resolution based on human cognition and consciousness. The introduction of such a model is in particular reasonable for networked co-operations, where conflicts can occur during the parallel preparation of a (partial) structural model (see scenarios in section 9.5).

For the CoBE project the so-called COBE AWARENESS FRAMEWORK has been proposed as a way for coordinating the activities within a networked co-operation. This framework realizes a *decentral* awareness model to coordinate the working activities within a networked co-operation. In contrast to other existing implementations of the awareness model, this framework does not rely on a global server that takes over the notification of users with awareness events. This decentral approach of the awareness framework respects the server-less and decentral topology of the DEEVOLVE architecture (and the underlying $SO_{P2P}A$ style). Each peer is not only capable of acting as a publisher of awareness events, but also as subscriber who receives events from other peers through *awareness channels*. At any time, a peer owner (representing a partner in the co-operation) can subscribe to other partners in order to become notified about events. More information on this framework can be found in [Alda *et al.*, 2006].

9.4.3 Peer Services for Construction: DocExchange

This section describes one of the peer services that have been developed for supplying the planning process carried out by structural engineers. The principle structure of this peer service is depicted in Figure 9-4.

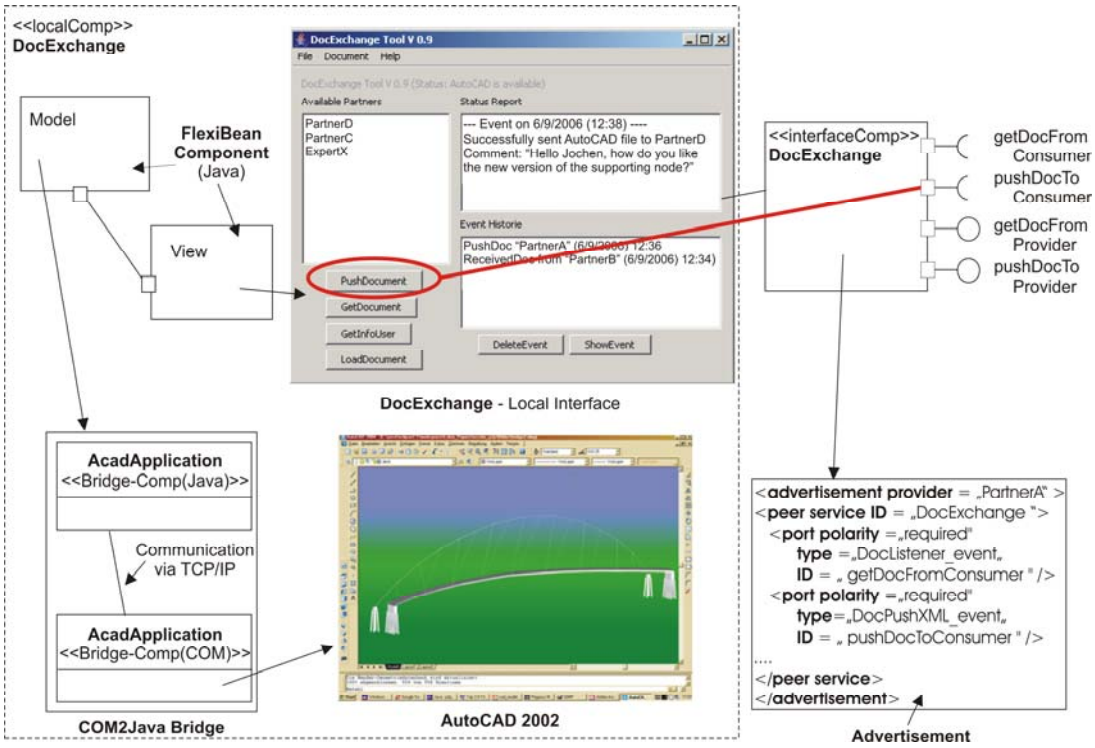


Figure 9-4: Structure of the "DocExchange" peer service

The "DocExchange" peer service enables engineers being members within a DEEVOLVE peer group to exchange design documents. Though it is possible to include any type of document from a technical perspective, the peer service is able to obtain documents that are available in the currently running AutoCAD installation.

The peer service essentially consists of two parts namely a local and an interface composition part. The local part implements a local administration interface (view component) and the bridge component to the local AutoCAD installation (model component using the COM2Java bridge, see section 9.4.1.2). The administration interface is a graphical user interface (Java Swing components) merely consisting of status panes (e.g. displaying the latest events in terms of a history) and a list giving an overview of currently connected partners. A button pane in the lower left corner enables a provider to send a document to a selected consumer (button "PushDocument") or to request a document from a selected partner (button "GetDocument"). For both operations, new dialog components are opened facilitating an engineer to enter auxiliary remarks that are also transferred to the dedicated partner. The request of a currently activated document as well as the display of a requested document in the AutoCAD installation is mastered by the COM2Java bridge. The prototypical implementation of this service uses AutoCAD version 2002. An upgrade to a newer version is certainly practical although not trouble-free as first attempts have shown.

The interface part consists of dedicated public ports allowing any remote peer to use the functionality of that peer service. Apart from some auxiliary ports (not mentioned here), the service realizes a bi-directional channel between the provider and the consumer allowing both to exchange documents in either ways. The ports "getDocFromConsumer" as well as "pushDocToConsumer" serve as the connection points for consumers allowing them to get documents or to receive requests for a document from the service provider. Both ports are activated after the provider has used the above-explained buttons "PushDocument" and "GetDocument", respectively (see red connection line in Figure 9-4). The service consumers themselves are able to send to or to

request documents from the service provider (ports “pushDocFromProvider” and “getDocFromProvider”).

The interface of the interface composition part is published within a service advertisement. Here, each port of the public interface is declared. Any peer is able to locate that advertisement and then to bind the “DocExchange” service. The consumer of course has to meet the peer group regulations being imposed by the provided peer service. A peer group could, for instance, describe any kind of logical unit such as a project in which a steel bridge construction (see scenario next section) is developed.

In principle, each application can use the interface of the DocExchange service. The composition of a local application together with (potentially many) remote DocExchange services is described as a PeerCAT composition. The local part of the “DocExchange” peer service also provides a further interface for binding remote “DocExchange” services provided from other peers with the local part. The “DocExchange” service then serves as a client and a server of AutoCAD documents at the same time. The list of currently available partners displays both provider of remote DocExchange services as well as consumers of that local peer service. Internally, the DocExchange service maintains a list indicating the role (consumer or provider) of a partner. Depending on the given role, a partner during an exchange operation is either addressed as a consumer (ports ending with suffix “Consumer”) or as provider (ports with suffix “Provider”). Owing to the capability of that service to serve as a client and a provider, transitive chains can be established between an arbitrary numbers of peers.

The local part of the “DocExchange” peer service can be connected with another remote peer service for performing a consistency check on a given document. Such a consistency service accepts an AutoCAD document as an input and returns a report as a result. This report describes the output of the executed consistency algorithm. A consistency checker could, for instance, evaluate a design document against modern fire protection rules. The project MADITA from the University of Darmstadt provides a useful implementation for checking fire protection rules in a design document (see [Theiss *et al.*, 2004] for more details). This module could necessarily be taken for an implementation of a “ConsistencyChecker” peer service. The invocation of the checker peer service can be triggered by pressing button “CheckDocument”. The “DocExchange” service fetches the recent document from the AutoCAD installation and passes it through the interface part of the “ConsistencyChecker” service to the local part of that service. Within the local part, the actual checker algorithm is carried out. Later, the local part sends back the result to the consuming service (here: the DocExchange service), where it is displayed accordingly.

9.5 Application Scenarios from Structural Design

This section presents exemplary structural design scenarios from the area of structural design in order to demonstrate the capabilities of the DEEVOLVE platform and its underlying models for exception handling and, to some extend, for dependency management. Here, a steel bridge construction has to developed, whereas the bridge accords to a real-world arched, pedestrian steel bridge that is located in Dessau, Germany (see [Alda *et al.*, 2006], [Bilek, 2006] for more details on this bridge and on the corresponding building project). All design scenarios depicted here conform to concrete scenarios that took place during the design and construction of that bridge, a couple of years ago. Background on those scenarios has been gathered by means of an

interview with a structural engineer, who was involved in that project[29]. At first, a general scenario is outlined that encompasses the basic set up of a networked co-operation. Then, two concrete scenarios are explained focusing on a useful application of both the adaptation policy and the integrity constraint concept.

9.5.1 Application Scenario "Set-Up of a Networked Cooperation"

The following general scenario is assumed for the steel bridge design scenarios:

Three different structural designers A, B, C and a project manager D (an architect) are willing to collaborate within a networked co-operation. Their aspired goal is to produce a coherent structural model of a steel bridge construction as a basis for future finite-element computations and structural detailing. The bridge's steel framework is composed of three main structural elements: i) the steel arch, spanning 108m with a slope of 17°, ii) 15 tension rods, that are associated with the arch and iii) the bridge deck composed of several steel panels. Four concrete abutments (two supporting the bridge deck, two supporting the arch) support the steel structure. The four abutments itself are founded on concrete piles.

The steel bridge necessitates the involvement of further yet unknown specialists (like expert X) for verifying complex abutments of that bridge. Here, deficient partial models are expected that could lead to subsequent modifications on all other partial models. Any additional expert should not stay in the group permanently. Three partners agree to work with AutoCAD 2002, partner A still works with AutoCAD 2000. All partners exchange their partial models w.r.t. IFC 2.2. All partners expect to be notified about changes on dependent partial models. All partners tend to work synchronously but at different locations. All partners identified common activities, in which the presence of at least a subset of partners is mandatory.

Realization in DeEvolve

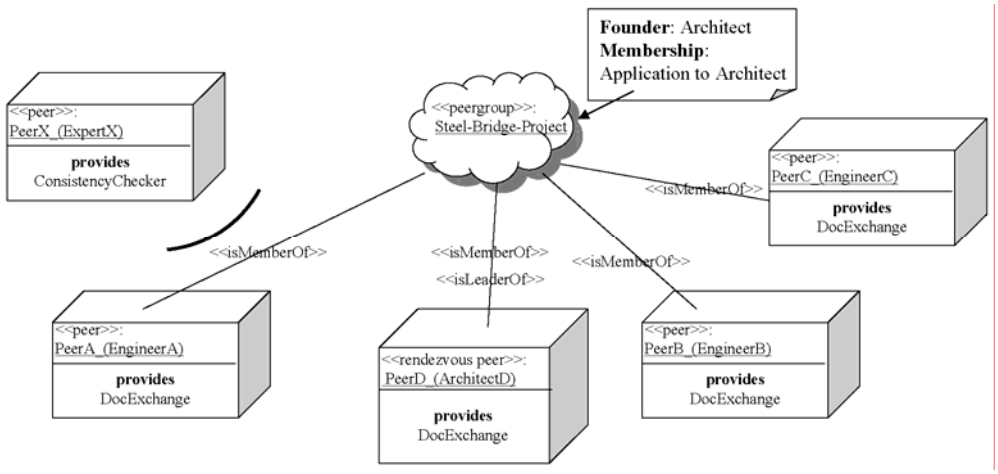


Figure 9-5: Structure of the DEEVOLVE architecture supporting the general application scenario of a networked co-operation

The structure of a distributed DEEVOLVE architecture supporting this use case is depicted in Figure 9-5. Each partner A - D within the co-operation is equipped with a DEEVOLVE peer environment that is to be installed on each terminal. The same termi-

[29] The interview was organized and conducted by Dr. Jochen Bilek, University of Bochum.

nal deploys the local AutoCAD 200X installation. The COM2Java bridge of DEEVOLVE allows to integrate AutoCAD into the peer environment. AutoCAD can now be published as a peer service to all other participants of the co-operation. From the given set of default peer services, for instance, the "DocExchange" service could be used to share AutoCAD-based documents (IFC) among the peers. A peer group represents the co-operation that defines the boundaries of it. This way, advertisements of peer services are only published within that peer group. The architect (peer D) serves as the founder and owner of that group. He authorizes new partners when they are willing to join the group. Only authorized members are able to access and bind a service in their environment. Thus, expert X first has to apply for membership before he can join the group. The peer of the architect takes over the role of a rendezvous peer serving as an internal, highly-available super peer within that given peer group.

9.5.2 Application Scenario "Handling Dynamic Availability"

The modeling of the northbound supporting node attaching the steel deck construction to the deck abutment involves two different partners (A, B) and an external expert (X) (see Figure 9-6 below). Partner A details the structural steel elements of the deck. Partner B designs the concrete abutment below the deck in this section. Certain subtle modifications of the steel deck (e.g. varying the height or width of the steel deck) could effect the reinforcement detailing of the underlying concrete abutment. On the other hand, new evaluation data concerning the soil condition of the ground (acquired by partner A) could not only influence the fundament of the abutment, but also effect the overlying steel deck (e.g. its weight). An external structural design expert (expert X) must assess each anxious modification on both models before subsequent modifications on dependent models are performed. Owing to the time pressure of the project, both assessment and execution of modifications should be handled synchronously without any serious delay. Architect D would like to be notified on potential changes.

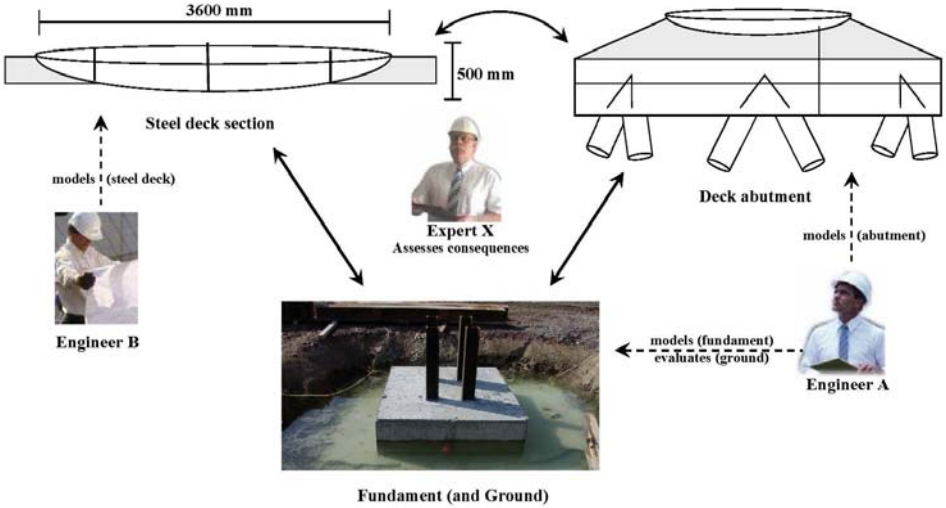


Figure 9-6: Visualization of scenario 1: involvement of three engineers during the parallel planning of a supporting node. The architect (project leader) is not depicted

Realization in DEEVOLVE

For supporting this subgroup of planners and expert(s), all affected users are bound among each other to exchange partial models and status information. Technically, this

can be achieved by locating and binding the respective peer service “DocExchange” provided by each partner into their local peer environments (see Figure 9-7).

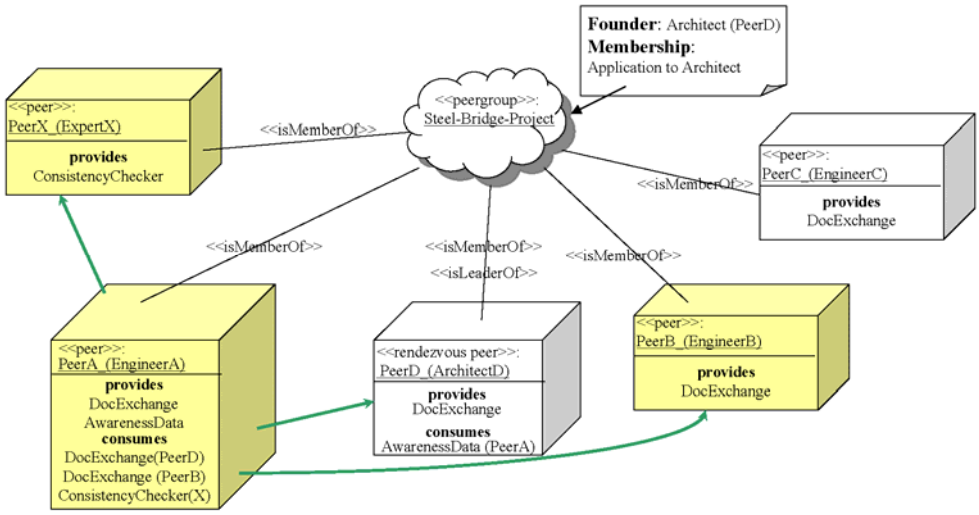


Figure 9-7: Structure of the DEEVOLVE architecture including the minimal composition during the modeling activity (shaded peers)

Furthermore, the peer environment of the external expert X must be included into the peer group. It is assumed that expert Y receives an invitation for joining that group. From a technical perspective, however, he must apply and join to that group, whereas the architect as the default project leader has to acknowledge his application. Then and only then the initial partners are able to bind the peer services of that expert into their environments. Expert X is supposed to provide a consistency checking service. Through this service, he is able to receive single design documents from the dedicated partners. Based on the obtained documents, expert Y can assess each modification, for instance, on models dealing with the steel deck.

```
<composition name = “ExtendedDocExchange" ID = “extendedDoc“ >
<services>
   <service name = “DocExchange "  ID = “remoteDataD“ host = “PeerD_(Architect)” >
   <service name = “DocExchange"  ID = “remoteDoc_B“ host = “PeerB_(EngineerB)” >
   <service name = “DocExchange"  ID = “localDoc_A“ host = “localpeer” >
   <service name = “ConsistencyChecker"  ID = “checkerX“ host = “PeerX_(Expert)” >
....
 <bindings>
    <rbind id="1">
       <port1 ID = “getDocFromConsumer " serviceID = “remoteDoc_B“ polarity = “required” />
       <port2 ID = “getDocFromConsumer " serviceID = “localDoc_A“ polarity = “required” />
    </rbind>
    <rbind id=“2”>
       <port1 ID = “pushDocToProvider " serviceID = “remoteDoc_B“ polarity = “provided” />
       <port2 ID = “pushDocToProvider " serviceID = “localDoc_A“ polarity = “required” />
    </rbind>
    ....
 </bindings>
</composition>
```

Figure 9-8: PeerCAT file of partner A defining the composition between partners A, B, D and Expert X (access ports of services are omitted)

The corresponding PeerCAT file from partner A defining the necessary composition between the pertaining partners is depicted in Figure 9-8. In this file, the “DocExchange*”* service of partner B is bound with the local “ExtendedDocExchange” application of partner A, so that models between these two partners can be exchanged. According to this example, partner A has located the advertisement of the “DocExchange” published by partner B and established a connection as a consumer. Along the

interconnected ports “getDocFromConsumer” (see binding statement in Figure 9-8), partner A is able to receive requests from partner B to send his current document. Alternatively, partner A can actively push the design model to the environment of partner B by invoking port “pushDocToProvider”. All other bindings have been omitted.

The involvement of expert X is indicated by the including service “Consistency-Checker”. The architect can be notified about ongoing changes within the environment of partner A by receiving awareness events through the service “AwarenessData” (see more information below) provided by partner A.

During the parallel modeling of the northbound supporting node, partners A and B as well as expert X have an immense responsibility within that co-operation. In order to guarantee the successful and correct progress of that particular modeling activity, these three engineers are supposed to establish a contract in terms of an integrity constraint. This constraint dictates that during the activity of modeling the northbound supporting node (= context), the involved peer services of the peers A, B and X must be available. Seen from the viewpoint of peer A, the services of peer B and X must be available during that context. This conforms to a minimal composition that must be fulfilled (see Figure 9-7 for a visualization). Figure 9-9 depicts the definition of the minimal composition of partner A. The condition is met, if the links to all peer services summarized in the semantics tag are available. The set of services defining the minimal composition is a true subset of services that is defined in the PeerCAT composition of Figure 9-8. The (only) context is defined as “Activity: Modeling the Northbound Supporting Node”. The engineer of peer A is able to set this context in his DEEVOLVE console, which activates the minimal composition with ID “activity1”.

```
<composition name = “ExtendedDocExchange" ID = “extendedDoc“ >
...
 <semantics>
    <constraint name = „Minimal_Composition“ ID = „ activity1” >
        <description> This contract has been established between all partners to ensure the
                      common modelling activity of the Northbound Supporting Node. </description>
        <integrityclass classname = „org.deevolve.integrity.MinimalComposition“ />
        <operation = „hasService“ >
            <params = „remoteDoc_B, localDoc_A, checkerX“ />
            <errorlevel value = „obligation“ description = “All services must be available” />
        </operation>
        <context value = „Activity: Modeling the Northbound Supporting Node“ />
    </constraint >
</semantics>
```

Figure 9-9: Declarative description of an integrity constraint in PeerCAT for a minimal composition

If the integrity constraint is fulfilled, all pertaining stakeholders are able to exchange data and information from their modeling activities in a reliable manner. This way, the consistency among all partial building models (here: models modeling the abutment and steel deck) can be ensured. In addition, additional value-added service can be bound and used steadily (here: the assessment of dependencies between the steel deck and the abutment by expert X).

The condition is violated if at least one peer service of the minimal composition becomes unavailable. In this case, each partner can choose among a fixed number of exception handlers that have been associated to that integrity constraint. Both exception handlers and the integrity constraint itself can be predefined by architect D and can be made available to all stakeholders. The exception handlers are defined in the same PeerCAT file that declares the integrity constraint and refer to its identifier (ID =

"activity1"). At first, all necessary actions are declared within the exception handlers tags that will be used later on in the options (Figure 9-10).

```
<composition name = "ExtendedDocExchange" ID = "extendedDoc" >
 ...
 <exceptionHandling>
  <actions>
    <action id ="discover" class="org.deevolve.integrity.Discover" description = "..." />
    <action id ="deploy" class="org.deevolve.integrity.Deployment" description = "..." />
    <action id ="bindService" class="org.deevolve.integrity.Bind" description = "..." />
    <action id ="subscribeToService" class="org.deevolve.integrity.Subscribe" description = "..." />
    <action id ="ignoreException" class="org.deevolve.integrity.Ignore" description = "..." />
    <action id ="switchToTailoring" class="org.deevolve.integrity.Tailoring" description = "..." />
    <action id ="applyMembership" class="org.deevolve.integrity.Membership" description = "..." />
    <action id ="cascadeException" class="org.deevolve.integrity.Cascade" description = "..." />
</actions>
```

Figure 9-10: Declaration of an exception handler in PeerCAT 1 (actions)

In the second part, all options are defined (see Figure 9-11). Options concretize the above-mentioned actions by passing concrete parameters to them. For instance, a concrete query string ("Consistency Checker Steel") is passed to the "discover" action. This string allows for seeking for a new peer service within a peer-to-peer architecture. The "bindService" action is passed concrete parameters indicating the required and the provided port from the new binding. Value "$serviceIDNew" is a variable, which is filled during runtime indicating the new identifier for the retrieved service. The action assumes that this parameter is classified and then passed by a previous action.

```
<exceptionHandling>
....
<handlers>
    <handler integrity = "activity1" >
      <options>
        <option id = "disOpt" action = "discover">
          <param name = "queryString" value = "Consistency Checker Steel"> ...
        </option>
        <option id = "deployServiceOpt" action = "deploy">
          <param name = "NewServiceName" value = "$serviceNameNew">
          <param name = "NewServiceID" value = "$serviceIDNew">
        </option>
        <option id = "bindOpt" action = "bindService">
          <param name = "NewServiceID" value = "$serviceIDNew">
          <param name = "Port1" value = "checkDocF">
          <param name = "LocalServiceID" value = "localDoc_A">
          <param name = "Port2" value = "checkDoc">
        </option>
        <option id = "subscribeOpt" action = "subscribeToService">
          <param name = "NewServiceName" value = "$serviceNameNew">
          <param name = "dep" value = "1"> ...
        </option>
        <option id = "ignoreOpt" action = "ignoreException">
        </option>
        <option id = "switchTailorOpt" action = "switchToTailoring">
          <param name = "ServiceComposition" value = "extendedDoc">
          <param name = "ToolID" value = "DeEvolveTailorTool"> ...
        </option>
        <option id = "cascadeOpt" action = "cascade">
          <param name = "Status" value = "Violated">
          <param name = "Text" value = "The Contract (Minimal Composition) has been violated">
        </option>
        <option id = "applyOpt" action = "applyMembership">
      </options>
```

Figure 9-11: Declaration of an exception handler in PeerCAT 2 (options)

In the third part, the above-declared options are arranged into concrete executable and selectable options (see Figure 9-12). By default, the violation of the integrity con-

straint is *automatically* delegated to all registered peers that have enabled the cascading of exceptions during subscription. In this application scenario, all members of the minimal composition are notified about the violation of the integrity constraint formulating the minimal composition. After the delegation of the exception to all dependent peers, the local user can select further options out of a list of selectable options.

```
...
    <handlers>
     <handler integrity = "activity1" >
      ...
      <automatically>
         <run option = "cascadeOpt"
            name = "Delegate Violation to Partners"
            description = "Delegates the violation of the constraints to all partners"/>
      </automatically>
      <manually>
        <select option = "disOpt, subscribeOpt, deployServiceOpt, bindOpt"
            name = "Locate new and trusted Consistency Checker in peer group"
            description = "Automatic Retrieval and Integration of new Consistency Checker Service.
                           Apply when Expert turns out to be unreliable. New Service must be in the same peer group
                           especially during engaged modeling activities " />
        <select option = "disOpt, subscribeOpt, switchTailorOpt"
            name = "Locate new and trusted Consistency Checker in peer group (Advanced Users)"
            description = "Semi-Automatic Retrieval and Integration of Consistency Checker Service.
                           For experts, when bindings are not known in advance />
        <select option = "disOpt, applyOpt, subscribeOpt, deployServiceOpt, bindOpt"
            name = "Locate New Consistency Checker"
            description = "Automatic Retrieval and Integration of new Consistency Checker Service.
                           Apply when Expert turns out to be unreliable. Can be out of the Peer Group in case
                           of relaxed project circumstances " />
        <select option = "ignoreOpt"
            name = "Ignore Violation of Constraints"
            description = "Ignores the violation in case when the checker service or the service of
                           partner is actually not needed. " />
      </manually>
     </handler>
    </handlers>
  </exceptionHandling>
...
<composition>
```

Figure 9-12: Declaration of an exception handler in PeerCAT 3 (executable and selectable options)

The selection of options depends on the actual state or progress of the given project context, in which the modeling activity is situated. The following justifications for each selectable option can be made:

- **Option "Locate new and trusted Consistency Checker in peer group":** This option features a sequence of four options, which is to be used when the consistency checker service of the expert X has caused the violation of the constraint with ID "activity1". The sequence starts by discovering a new consistency checker service ("disOpt"). Within this option, the user can determine the most suitable service among the retrieved ones. Afterwards, the peer subscribes as a consumer of the chosen service into the DEEVOLVE environment of the respective service provider ("subscribeOpt"). The service is then integrated by deploying the service ("deployServiceOpt") and by binding the respective ports of the new service with the existing local "DocExchange" application ("bindOpt"). This sequence of four options assumes that the service is provided by a *trusted* service provider within the same peer group representing the current collaboration. This option should be applied during busy modeling activities, where the expert turns out to be unreliable though his presence is mandatory. It is practicable in situations, in which there is no room left for trusting or testing external services outside established peer groups. If in fact no peer service has been found, option "disOpt" displays the unsuccessful attempt to locate a service to the user and aborts the sequence (see sec-

tion 8.4.4., last paragraph). In this case, the handler dialog is displayed again allowing the user to choose a new option.

- **Option "Locate new and trusted Consistency Checker in peer group (Advanced Users)**" This option has the same assumption as the previous option but involves the local user to deploy and to bind the service into the running application ("switchToTailor"). This option is useful especially for advanced users. It is also reasonable to select this option in the case when interface ports of the new service are not completely known at design time of the handlers.
- **Option "Locate New Consistency Checker":** This option also presumes that the peer of an expert has become unavailable, which then causes the violation of the constraint. During stable project circumstances (e.g. at the beginning of a project), the user could consider to integrate external, untrusted service providers. In the case that such a service is outside of the established peer group, an additional option "applyOpt" has been inserted for applying the membership to that group. If the application has been granted by the group leader, the process of integrating the service continues in the same way as in the first option. The option may take an unpredictable long time, if the application is not confirmed by the group leader.
- **Option „Ignore Violation of Constraints"**: This option allows for ignoring the violation of the constraints. It can be applied, if the violation is not relevant at a given point in time. The option can be taken in situation when either the expert or the associated partner becomes unavailable.

During deployment, DEEVOLVE interprets and transforms these options into executable Java code. Given the violation of an integrity constraint, a dialog is presented to the partner who has detected the violation (Figure 9-13).

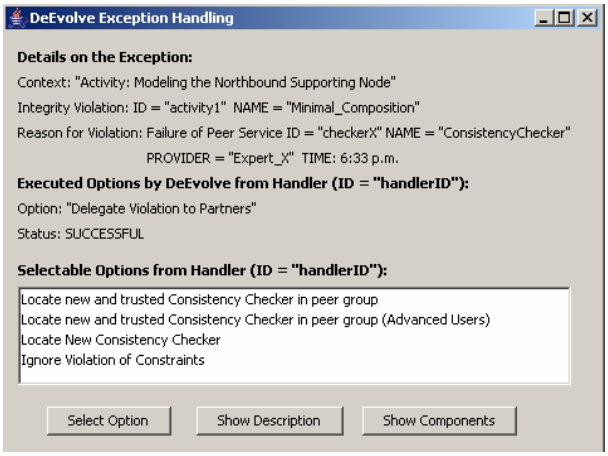


Figure 9-13: Dialog for selecting options from an exception handler

In this dialog, the *id* and the *type* of the violated integrity constraint are displayed as well as an indication of the causing reason (here: the failure of peer service provided by expert X). Based on this information, the partner can choose among the four established handlers as previously defined in the corresponding PeerCAT description. The dialog also displays, which option has been executed autonomously by the DEEVOLVE environment (here: the cascading of the occurred violation to all subscribed partners).

Additional Support: The CoBE Awareness Framework

Besides the ability to control the availability of partners and to define integrity constraints on co-operations, partners may fall back on the COBE AWARENESS FRAMEWORK to perceive concrete modeling activities among each other. To subscribe into

the awareness channel of a partner, a consumer is asked to locate and to use the peer service "AwarenessData" of the respective partner and to register as an interested user. Then, the user will be notified about important changes on design documents, saving and opening of documents and so on. Again, the capability of DEEVOLVE to monitor the peer group's availability (state) is also profitably for the awareness framework. The synchronous collaboration of partners by means of the awareness framework can only be established if the peers are reachable. The concrete possibilities of the awareness framework can be looked up elsewhere (e.g. [Alda, 2005b]). In the given scenario, the architect uses the peer service "AwarenessData" of partner A to become notified about his activities (e.g. the closing of document, the change of a model).

9.5.3 Application Scenario "Handling Adaptation Requests"

"*During the last third of the project, partner A decides to upgrade his local AutoCAD 2000 installation to version 2003. Obviously, this upgrade promises an improvement of his working environment. Partner A would like to ensure that this upgrade does not violate any contextual dependencies within the given co-operation (peer group).*"

Realization in COBE and DEEVOLVE

Indeed, the update as described in the second scenario leads to various problems. The COBE AWARENESS FRAMEWORK makes use of the COM-based event channels of AutoCAD to determine incidents on modifying documents and to fetch a copy of the current design model. In newer AutoCAD versions (from 2003), however, the COM interface has been modified extensively. So, composing newer AutoCAD versions with our awareness framework is no longer feasible. Consequently, partners of the peer group could no longer be notified about modifications on design models. Besides this software compatibility concern, the leader of the peer group (architect) may be worried about the compatibility of the exchange formats. Although all *newer* AutoCAD versions apply the IFC standard, incompatibilities may occur due to improper implementations especially of rarely used aspects. The availability of incompatible documents is a problem when merging the partial models to a complete model.

The DEEVOLVE platform utilizes the notion of adaptation policy to avoid potential risks when adapting local software. In this scenario, the architect acting as the group founder is responsible to define and to provide the adaptation policy for his group. The selected adaptation strategy depends on the current project status. Towards the end of a (critical) project, the architect could update the policy imposing a more restrictive dealing with adaptation requests. A possible strategy could entail that each adaptation request should be negotiated with all partners of the group. An extension could state that, given at least one strong dependency on a service, no adaptation is allowed.

9.6 Conclusion

This section has shown possible application scenarios stemming from the area of construction engineering in order to demonstrate the capabilities of the DEEVOLVE runtime environment. The focus of this chapter has been to point out the particular strength of DEEVOLVE to support the collaboration of dispersed working people in a scenario of unreliable und fluctuating partners. Here, the notions of exception handling and integrity constraints have turned out to be beneficial. The demonstration of DEEVOLVE's consumer dependencies mechanism has been illustrated reasonably.

Chapter 10
Conclusion and Future Work

This chapter presents the concluding remarks of this dissertation. At first, section 10.1 summarizes core *contributions* of this work. Subsequently, section 10.2 presents critical remarks that outline the *limitations* of this work. Section 10.3, finally, offers a visionary view of potential future work based on the findings presented in this work.

10.1 Contributions

This thesis has provided several basic contributions for enhancing the adaptability of service-oriented peer-to-peer architectures. The initial research questions stated in chapter 1 are here addressed again in order to sum up the contributions of this work. It will be shown to what extent these questions have been answered in the last chapters:

Semantics of Service-Oriented Peer-to-Peer Architecture (mainly addressing research question Q1, also Q2 and Q3): The semantics of a service-oriented peer-to-peer architecture have been formulated in two ways. At first, an architectural style ($SO_{P2P}A$) based on a process calculus has been introduced in section 3 in order to rigorously formalize the fundamental elements of such an architecture, their correlations, and their constraints. The selected style has demonstrated particular strength in formalizing *deployable processes* (e.g. components, services)*, environment processes* (e.g. peers, peer groups)*,* and corresponding *manipulation concepts* (e.g. adaptation of processes). In contrast, the *diagnostic concepts* (e.g. exception detection, dependency analysis, maintenance of integrity constraints) have been formalized only rudimentarily. Here, the architectural style turned out to be rather uncomfortable. The style proposed to *substantiate* these concepts for a concrete realization of a service-oriented peer-to-peer architecture covering the circumstances of a concrete application domain (see justification in sections 4.2 and 4.3). In fact, these concepts are presented in more depth during the presentation of the DEEVOLVE environment in Chapters 7 and 8.

Component-based Adaptation Methods for End-User Tailoring (Q2) and for Exception Handling (Q3), Component-based Decomposition Model (Q4). This thesis proposed the adoption of *component-based adaptation methods* for tailoring peer services and service composition. These methods enable users to tailor services and service compositions in terms of intuitive operations, such as adding components or services, or deleting bindings. For a service-oriented peer-to-peer architecture, the original set of component-based adaptation methods as suggested by Stiemerling [Stiemerling, 2000] and Won [Won, 2004] has been supplemented by additional methods. These methods incorporate the conditions of a service-oriented peer-to-peer archi-

tecture (e.g. discover a new service, subscribe to a peer service, notify and involve users). The same adaptation methods have been taken as actions to handle the occurrence of an exception (i.e. the loss of a peer service), as well as the violation of an integrity constraint. Users who are familiar with tailoring software will probably have minor problems with creating exception handlers or with handling an exception during runtime. For realizing these adaptation methods, peer services are modeled as component-based peer services, that is, as compositions of components. Service compositions feature the same interaction principles and binding concepts as component compositions. Thus, adapting service compositions can also be mastered by the same component-based adaptation methods.

The *cost efficient development* of services and service-oriented applications capable of handling exceptions is guaranteed by the novel approach to exception handling as proposed in this dissertation. Here, the *runtime environment* implements all the code necessary for handling exceptions. The flexible broker-based approach (section 8.3.2) guides the remote interaction between peers and is able, if necessary, to trigger the process of exception resolution. In application scenarios with complex and unpredictable context information, the user can be involved in selecting appropriate handlers at runtime. Thus, the developer of a peer service needs not care about any code for handling exceptions. Aspects relating to *runtime-efficiency* have only been evaluated regarding the implementation of the broker approach. The generation of the necessary broker produces a critical slow-down of the overall deployment process (see section 8.3.2). However, since brokers are only instantiated once during the first start, this slow-down can be accepted. Improvements have been proposed, but have not been implemented so far.

Trade-Off between User Involvement and an Adaptive Environment, Interpretation of Context Information (Q5, Q6). As mentioned before, users can be actively involved during the process of exception handling. The flexible way of arranging exception handlers (section 8.4.1) allows for varying the degree of user involvement. In principle, a purely adaptive environment can be configured in which all options are executed autonomously. This work has presented an application scenario in which user involvement turns out to be more effective due to the complex context data available in these scenarios. One implication of this thesis is that the recognition and interpretation of such context data needs to be mastered by the end-users themselves. Based on this interpretation, appropriate options for handling exceptions can then be selected.

Description and Maintenance of Dependencies (Q7): From the viewpoint of a single DEEVOLVE peer environment, two types of dependencies must be maintained, namely dependencies on provided (used) peer services and dependencies of consumer peers replying to a local service. Dependencies on provided services are explicitly described by the binding statements in a PeerCAT service description. These dependencies can be further enriched by dependency values denoting the relevance of a service. The subscription to peer services facilitates a peer to register as a consumer of a service with the corresponding provider peer. The provider peer can use this data to analyze consumer dependencies prior to the adaptation of a peer service. The structure of the “PeerStore” table in a local peer environment makes it possible to derive transitive dependencies. This is in particular important during the process of delegating the event of an exception to dependent peers (exception cascading, see section 8.3.5).

Respecting Consumer Dependencies in a Scalable Way (Q8): In order to address this problem, the notion of a peer group-based *adaptation policy* has been proposed.

Instead of arranging single consumer-provider policies (which is clearly not scalable with a growing number of peers), peer groups agree to a common adaptation policy during their set up. New peers joining these peer groups have to comply with these pre-defined policies. These describe exact procedures of how to cope with existing consumer dependencies.

Reliability Collaboration on an Architectural Level (Q9): In order to face this concern, the notion of *integrity constraints* has been suggested. These constraints correspond to so-called *contracts* that can be negotiated between providers and consumer, and make assumptions on the availability of services in dedicated working contexts. The exception handling approach is adopted for detecting and handling the violation of these contracts.

10.2 Limitations of the Contributions

The contributions of this thesis as presented above exhibit a number of limitations or open issues that have been recognized, but not solved in the course of this research project. Some of these limitations naturally offer implications for future work for this dissertation project. The following limitations are obvious:

Weakness of the JXTA framework: Many problems have emerged during the use of the JXTA framework. Although the concepts of JXTA (including the protocol specifications) are certainly well-defined, the (only available) prototype implementation of that framework is rather weak. A critical and somewhat unreliable building block of this framework constitutes the process of discovering service or group advertisements, especially when the peers are located behind a firewall. Even in rather simple settings (e.g. two laptops connected to the same LAN), published advertisements could often not be mutually discovered. The peer group membership protocol, which is available to regulate group memberships, also does not function satisfactorily, since the process e.g. for an application takes an unpredictably long time. Moreover, the different versions of the framework have no backward compatibility. The weakness of JXTA has been mastered by some work-around implementations like further alternative communication protocols (e.g. the DEEVOLVE messaging service 6.6.4, which is based on the Java Mail framework). Future implementations should definitely fall back on modern Web Service-based frameworks (e.g. AXIS-2 by Apache).

Complexity of FREEVOLVE: The initial structural model of a distributed application FREEVOLVE (cf. Figure 5-3 and Figure 5-4) has turned out to be complex to grasp especially for novice users and students. Consequently, the structural model of a distributed application in DEEVOLVE (section 6.3.4) has become complex as well. Most students have evaluated the component-proxy structure (used for regulating the adaptation process of components) as tedious and hard to understand. For further development of DEEVOLVE, the structural model must perhaps be refined towards a more transparent model, for instance by reducing the number of used concepts (i.e. classes).

Small Number of Tools for DEEVOLVE: For both the composition and the adaptation of peer services, only few tools have been implemented. Thus, the composition of peer services has to be mastered mainly with the textual editor of DEEVOLVE (section 6.6.5). Although practicable, the process of composition is rather cumbersome, especially for unskilled users. The development of more tools for the adaptation of peer services might potentially have served as a starting point for identifying new ergo-

nomic design principles for such tools. Tools incorporating such (established) guidelines would also increase the users' acceptability of the DEEVOLVE architecture.

PeerCAT Language does not rely on Standards: The PeerCAT composition language is not based on standard notations for Web service composition such as BPEL4WS or BPEL. Although these standards merely describe workflow compositions rather than structural compositions, some syntactical aspects could have been adopted (e.g. exception handling). The compatibility of (or the similarity to) composition standards from the area of Web service would certainly increase the *portability* of the essential contributions of this thesis (e.g. user-oriented exception handling, integrity constraints) to that area.

No Rollback Mechanism: An appropriate rollback mechanism as mentioned in section 4.4 would improve the reliability of runtime adaptation of an instance of a peer service or composite peer service. Again, a rollback mechanism could ensure that *all* instances of a peer services are adapted consistently. In case of the failure of a consumer peer that is one of the peers relying on an instance of the adapted service, the adaptation process would be rejected (i.e. rolled backed). This way, *no* instance of a peer service would be adapted, which would, however, lead to a consistent state.

No Recommendations for the Integration of DEEVOLVE in existing Collaborations: This work has not provided any analysis of how to integrate DEEVOLVE (together with appropriate services) into an existing *real-world* collaboration. As is known from other studies that have evaluated the introduction of software systems into existing organizations, developers have to expect *barriers* from the participants to accepting the new technology. Therefore, appreciated methods for the introduction of software into an existing group (e.g. ethnographic methods [Hughes *et al.*, 1994]) are necessary. Again, the above-mentioned software ergonomic guidelines could be useful for increasing the acceptability of tools and services in such collaborations.

10.3 Outlook on Future Work

This section envisions four distinguishing directions of future work that could follow and, thus, enrich the findings of this dissertation project.

Portability of Findings to Web Service Architectures: Towards the end of this project, it has turned out that the Web Services stack (SOAP, WSDL, UDDI, and BPEL for service composition) constitutes the state-of-the-art technology framework for implementing service-oriented architectures. For this framework, the three most relevant contributions of this thesis (user involvement during exception handling, consumer analysis based on flexible adaptation policies, as well as integrity constraints for realizing service contracts) have not been addressed. To date, the most dominant application field of Web services is the integration of various applications within an enterprise towards a coherent, workflow-based application (EAI = Enterprise Application Integration). Here, aspects of user involvement are not that relevant, at first sight. The contributions of this work mentioned before will become relevant if Web Services are deployed in more user-involving application scenarios for supporting the collaboration of users and user groups. Especially in workflow-based scenarios, user involvement could be beneficial to handling exceptions that occur during the execution of a workflow. Here, the implementation of purely adaptive mechanisms for solving such exceptions would also result in too many complex solutions.

Analysis of Tailorability in Workflow-based (Web Service) Compositions: The concept of component-based tailorability as promoted by research conducted by Stiemerling, Won, as well as by this dissertation project relies on structural composition. Here, somewhat easy and intuitive adaptation methods can be described that can also be implemented effectively in an adaptation environment. As outlined in section 2.5.3.1, new languages and engines for service composition (from the SOA context) rely on workflow-based models. The analysis of end-user tailorability of these workflow-based composition approaches is clearly a new research field. In this light, it has to be evaluated how important requirements of tailorability (such as the runtime affection of adaptation steps to composition instances) can be integrated in runtime engines. Moreover, research could be carried out in order to *formulate* adaptation methods based on the typical elements of workflow languages (including forks, condition, etc.). These new methods need to be evaluated in concrete scenarios in order to see how users cope with them. It can be expected that these adaptation methods will become more complex than component-based methods based on structural composition.

Advanced Concepts for Modeling and Maintaining Context Information in Integrity Constraints: The context in an integrity constraint indicates a situation (e.g. a planning activity or a certain time span) in which a given constraint has to be satisfied. Here, users can switch between many pre-defined contexts, which in turn activate the pertaining integrity constraints. The context model can be extended and improved in many ways. A *context taxonomy* could be established in order to describe the relationship among many contexts and, thus, to structure all available contexts. Such a taxonomy could be described by means of conventional class diagrams or even ontology-based languages and notations. For example, an appropriate notation could allow for the description of context hierarchies according to the generalization/specialization pattern of the object-oriented paradigm. The runtime engine could use this hierarchy in order to evaluate context information. For instance, if a single context is chosen by a user and fixed within the DEEVOLVE environment, then not only the directly associated integrity constraints must be satisfied, but also all constraints that belong to specializations of that single context with respect to a given context taxonomy.

A further improvement would be for users within a peer group to be capable of *perceiving context* changes of other users. Again, a context change aims at indicating a transition to another working activity. A local context transition could be a useful piece of information for other users who potentially work in the same project or on the same resources, respectively. As a suitable reaction, these users could also adapt their current context information and, thus, activate the corresponding integrity constraint. This approach could incorporate concepts and mechanisms of the well-known awareness model [Dourish and Bellotti, 1992] stemming from the discipline of Computer Supported Cooperative Work (CSCW). An appropriate approach would clearly improve the collaboration of all users who work to achieve a common goal.

Reputation as a Way to assess User Behavior: A reputation model has been proposed in the formal $SO_{P2P}A$ architectural style (section 4.1.2), but has not been implemented in DEEVOLVE. The overall benefit of such a reputation model is its ability to support the service consumer's attempt to determine and bind reliable peer services. With respect to the original idea of $SO_{P2P}A$, a reputation model should allow any user to formulate reputation values and to publish them to all other users (e.g. as an advertisement, or into a central directory). Users could assess service providing peers with a negative reputation value after the careless or defective adaptation of a published peer service that has violated the convention of an adaptation policy. The (permanent) vio-

lation of an established integrity constraint that serves as a contract within a user collaboration could also diminish the reputation of the respective peer owner. A reputation model is suitable for creating new collaborations that aim at involving trustworthy parties only. Apart from the implementation work, a sound formal reputation model has to be chosen from typical state-of-the-art models (see 2.2.3 for an overview).

References

[Acquisti *et al.*, 2003] Acquisti, A., Dingledine, R. and Syverson, P.: "On the Economics of Anonymity. Financial Cryptography", in: Proceedings of the *Seventh International Financial Cryptography Conference*. Gosier, Guadeloupe. 2003.

[Alda, 2002] Alda, S.: "Agentenbasiertes Modell zur Geschehens- und Zustandswahrnehmung im Bauwesen", in: Proceedings of the *14th Forum Bauinformatik.* University of Bochum, Germany. September 2002.

[Alda, 2004] Alda, S.: "Component-based Self-Adaptability in Peer-to-Peer Architectures", in: Proceedings of the *26th International Conference on Software Engineering (ICSE 2004). Contribution for the Doctoral Symposium*. Edinburgh, Scotland. May, 2004. (pp. 33-35).

[Alda, 2005a] Alda, S.: "Peer Group-Based Dependency Management in Service-Oriented Peer-to-Peer Architectures", in: Proceedings of the *Third International Workshop On Databases, Information Systems and Peer-to-Peer Computing (DBISP2P), in conjunction with 31st International Conference on Very Large Data Bases (VLDB)*. Trondheim, Norway. September 2005.

[Alda *et al.*, 2006] Alda, S., Bilek, J., Cremers, A. B. and Hartmann, D.: "Awareness and workflow based coordination of networked co-operations in structural design", in: *Journal of Information Technology in Construction (ITCon), Special Issue Process Modelling, Process Management and Collaboration,* Vol. 11, No. July 2006. (pp. 489-507).

[Alda and Cremers, 2004] Alda, S. and Cremers, A. B.: "Strategies for Component-based Self-Adaptability Model in Peer-to-Peer Architectures", in: Proceedings of the *4th International Symposium on Component-based Software Engineering (CBSE7)*. Springer (LNCS 3054). Edinburgh, Scotland. May 2004. (pp. 59-67).

[Alda and Cremers, 2005] Alda, S. and Cremers, A. B.: "Towards Composition Management for Peer-to-Peer Architectures", in: *Workshop Software Composition (SC 2004), affiliated to the 7th European Joint Conference on Theory and Practice of Software (ETAPS 2004). Published in Electronic Notes in Theoretical Computer Science,* Vol. 114, No.January 2005. 2005. (pp. 47-64).

[Alda *et al.*, 2004] Alda, S., Cremers, A. B., Bilek, J. and Hartmann, D.: "Support of Collaborative Structural Design Processes through the Integration of Peer-to-Peer and Multiagent Architectures", in: Proceedings of the *10th International Conference on Computing in Civil and Building Engineering (ICCCBE-X)*. Weimar, Germany. June 2004.

[Alda and Mitrov, 2004] Alda, S. and Mitrov, T.: "Semantic Integrity Concepts for Service-Oriented Peer-to-Peer Architectures", in: Proceedings of the *Metadata Management in Grid and P2P Systems (MMGPS)*. London, England. December 2004.

[Alda *et al.*, 2002a] Alda, S., Radetzki, U., Bergmann, A. and Cremers, A. B.: "A Component-Based and Adaptable Platform for Networked Cooperations in Civil and Building Engineering", in: Proceedings of the *9th International Conference on Computing in Civil and Building Engineering (ICCCBE-IX)*. Taipei, Taiwan. April 2002.

[Alda *et al.*, 2002b] Alda, S., Radetzki, U., Won, M. and Cremers, A. B.: "Eine anpassbare, komponentbasierte Plattform für vernetzte Kooperationen im Bauwesen", in: Proceedings of the *Tagung Bauen mit Computern der VDI*. Bonn, Germany. April 2002. (pp. 351-370).

[Alda *et al.*, 2002c] Alda, S., Won, M. and Cremers, A. B.: "Managing Dependencies in Component-Based Distributed Applications", in: Proceedings of the *2nd International Workshop on Scientific Engineering of Distributed Java Applications (FIDJI'2002)*. LNCS 2604, Springer. Luxembourg-City, Luxembourg. November 2002.

[Alda, 2005b] Alda, S. C., A.B., Bilek, J., and Hartmann, D.: "Integrated Multiagent and Peer-to-Peer based Workflow-Control of Dynamic Networked Cooperations in Structural Design", in: Proceedings of the *22nd International Conference Information Technology in Construction (CIB-W78).* Dresden, Germany. Juli 2005.

[Allen *et al.*, 1997] Allen, R., Douence, R. and Garlan, D.: "Specifying Dynamism in Software Architectures", Proceedings of Workshop", in: Proceedings of the *Foundations of Component-Based Systems*. Zurich, Switzerland. September 1997.

[Amadio, 2000] Amadio, R. M.: "On modeling mobility", in: *Theoretical Computer Science,* Vol. 240, No.1. June 2000. (pp. 147-176).

[Arnold *et al.*, 1999] Arnold, K., O'Sullivan, R., Scheifler, W. and Wollrath, A.: *The Jini Specification.* Addison-Wesley, Reading, Mass. 1999.

[Badr *et al.*, 2002] Badr, N., Reilly, D. and Taleb-Bendiab, A.: "Conflict Resolution Control Architecture for Self-Adaptive Software", in: Proceedings of the *International Workshop on Architecting Dependable Systems: WADS 2002*. 2002.

[Barbosa, 2000] Barbosa, L. S.: "Components as processes: An exercise in coalgebraic modeling (preprint)", in: Smith, S. F. and Talcott, C. L. (eds.): *FMOODS'2000: Formal Methods for Open Object-Oriented Distributed Systems.* Kluwer Academic Publishers. Stanford, USA. 2000. (pp. 397--417).

[Barkai, 2002] Barkai, D.: *Peer-to-Peer Computing. Technologies for sharing and collaboration on the Net.* Intel Press. 2002.

[Barnatt, 1995] Barnatt, C.: "Office Space, Cyberspace and Virtual Organization", in: *Journal of General Management,* Vol. 20, No.4. 1995. (pp. 78-91).

[Bass *et al.*, 2003] Bass, L., Clements, P. and Kazman, R.: *Software Architecture in Practice.* Addison-Wesley. 2003.

[BEA *et al.*, 2003] BEA, IBM, Microsoft and SAP: *Business process execution language for Web Services (BPEL4WS), Version 1.1*. 2003. (ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf)

[Benatallah *et al.*, 2005] Benatallah, B., Dijkman, R. M., Dumas, M. and Maamar, Z.: "Service Composition: Concepts, Techniques, Tools and Trends", in: Stojanovic, Z. and Dahanayake, A. (eds.): *Service-Oriented Software System Engineering: Challenges and Practices.* IDEA Group Publishing. London. 2005.

[Ben-Shaul *et al.*, 2000] Ben-Shaul, I., Gazit, H., Holder, O. and Lavva, B.: in: Proceedings of the *First international workshop on Self-adaptive software*. ACM Press. 2000. (pp. 134-142).

[Bialek and Jul, 2004] Bialek, R. P. and Jul, E.: "A framework for evolutionary, dynamically updatable component-based systems", in: Proceedings of the *24th IEEE International Conference on Distributed Computing Systems Workshops*. Hachioji, Tokyo, Japan. March 2004. (pp. 326-331).

[Bilek, 2006] Bilek, J.: *Agentenbasiertes Kooperationsmodell zur Unterstützung vernetzter Planungsprozesse in der Tragwerksplanung.* Dissertation, Institute for Computational Engineering, University of Bochum, Germany. Bochum, Germany. 2006.

[Birrell *et al.*, 1982] Birrell, A. D., Levin, R., Needham, R. M. and Schroeder, M. D.: "Grapevine: An Exercise in Distributed Computing", in: *Communications of the ACM,* Vol. 25, No.4. 1982. (pp. 260-274).

[Bisignano *et al.*, 2003] Bisignano, M., Calvagna, A., Di Modica, G. and Tomarchio, O.: "Experience: a Jxta middleware for mobile ad-hoc networks", in: Proceedings of the *International Conference on Peer-to-Peer Computing (P2P2003)*. Linköping, Sweden. 2003. (pp. 214-215).

[Bode *et al.*, 2004] Bode, T., Devooght, I., Kolbe, T., Steinrücken, J. and Won, M.: "GeoCafé - Kommunikationszentriertes Gruppenlernen von Methoden der raumbezogenen Datenverarbeitung", in: Plümer, L. and Asche (eds.): *Geoin-*

formation - Neue Medien für eine neue Disziplin. Wichmann, Heidelberg. 2004.

[Booch *et al.*, 2005] Booch, G., Rumbaugh, J. and Jacobsen, I.: *The Unified Modeling Language User Guide, 2nd edition.* Addison Wesley. 2005.

[Borgström, 2003] Borgström, J.: *Process Calculi for the Foundations of Peer-to-Peer Systems. A Research plan.* Technical Report of L'Ecole Polytechnique Fédérale de Lausanne (EPFL). http://lampwww.epfl.ch/~jobo/Publications/Bor03.pdf. 2003.

[Borgström *et al.*, 2004] Borgström, J., Nestmann, U., Alima, L. O. and Gurov, D.: "Verifying a Structured Peer-to-peer Overlay Network: The Static Case", in: Proceedings of the *Global Computing 2004*. Springer (LNCS 3267). 2004. (pp. 251-266).

[Boudol, 1992] Boudol, G.: *Asynchrony and the pi-calculus.* Technical Report, RR1702, INRIA. Sophia-Antipolis. France. 1992.

[BPMI, 2003] BPMI: *Business Process Modelling Language (BPML) - Specification*. Business Process Management Initiative (BPMI). 2003. (http://www.bpmi.org/bpml-spec.htm)

[Brookshier *et al.*, 2002] Brookshier, D., Govoni, D. and Krishnam, N.: "JXTA: Java P2P Programming", in: (eds.): SAMS. Indianapolis. 2002.

[Bruegge and Dutoit, 2004] Bruegge, B. and Dutoit, A. H.: *Object-Oriented Software Engineering: Using UML, Patterns, and Java.* Pearson Prentice Hall, London. 2004.

[Buschmann, 1996] Buschmann, F.: *Pattern-Oriented Software Architecture, Vol.1: A System of Patterns.* John Wiley and Sons Ltd. 1996.

[Bussler, 2003] Bussler, C., Maedche, A., and Fensel, D.: "Web Services Quo Vadis?" in: *IEEE Intelligent Systems, Trends and Controversies,* Vol. 2003, No.January/February. 2003.

[Cardelli, 1997] Cardelli, L.: "Type Systems", in: Tucker, A. B. (eds.): *Computer Science and Engineering Handbook.* CRC Press, ACM. 1997. (pp. 2208--2236).

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P.: "On Understanding Types, Data Abstraction, and Polymorphism", in: *ACM Computing Surveys,* Vol. 17, No.4. 1985. (pp. 471--522).

[Carr, 1991] Carr, J.: "SNA and LU6.2 Connectivity", in: *LAN Network Magazine,* Vol. No.February 1991. 1991.

[Cervantes and Hall, 2004] Cervantes, H. and Hall, R. S.: "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model", in: Pro-

ceedings of the *26th International Conference on Software Engineering (ICSE 2004)*. Edinburgh, Scotland. May 2004.

[Cervantes and Hall, 2005] Cervantes, H. and Hall, R. S.: "Technical Concepts of Service Orientation", in: Stojanovic, Z. and Dahanayake, A. (eds.): *Service-Oriented Software System Engineering: Challenges and Practices.* IDEA Group Publishing. 2005.

[Cheng *et al.*, 2002] Cheng, S.-W., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B. and Steenkiste, P.: "Exploiting Architecture Style for Self-Repairing Systems", in: Proceedings of the *Proceeding of the International Conference on Software Engineering (ICSE)*. Orlando, USA. May, 2002.

[Chothia and Stark, 2001a] Chothia, T. and Stark, I.: "A Distributed pi-calculus with local area of communication", in: *Electronic Notes in Theoretical Computer Science,* Vol. 41, No.2. 2001.

[Christian, 1995] Christian, F.: "Exception Hanndling and Tolerance of Software Faults", in: Lyu, M. (eds.): *Software Fault Tolerance.* Wiley Publishing. 1995.

[Church, 1941] Church, A.: *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, N.J. 1941.

[Costanza, 2004] Costanza, P.: *Transmigration of Object Identity, Dissertation.* Dissertation, Institut für Informatik III, Universität Bonn. Bonn, Germany. 2004.

[Cremers and Alda, 2004] Cremers, A. B. and Alda, S.: *Komponentenbasierte Plattform für anpassbare, vernetzte Systeme im Bauwesen. Zwischenbericht über die 2. Phase im Rahmen des DFG-Schwerpunktprogramms SPP 1103.* Bonn, Germany. 2004.

[Cremers and Hibbard, 1978] Cremers, A. B. and Hibbard, T. N.: "Formal Modeling of Virtual Machines", in: *IEEE Transactions on Software Engineering,* Vol. SE-4, No.5. September 1978. (pp. 426-436).

[Cremers and Hibbard, 1986] Cremers, A. B. and Hibbard, T. N.: "Subspaces: Factorization and Communication", in: Byrnes, C. I. and Lindquist, A. (eds.): *Computational and Combinatorial Methods in Systems Theory.* Elsevier Science Publishers. Holland. 1986.

[Crowcroft *et al.*, 2004] Crowcroft, J., Moreton, T., Pratt, I., Twigg and A.: "Peer-to-Peer Technologies", in: Foster, I. and Kesselmann, A. (eds.): *GRID2 Blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers. 2004. (pp. 593-622).

[Curbera *et al.*, 2000] Curbera, F., Weerawarana, S. and Duftler, M. J.: "On Component Composition Languages", in: Proceedings of the *Fifth International Workshop on Component-Oriented Programming (WCOP 2000) in conjunction with ECOOP 2000*. 2000.

[Dashofy *et al.*, 2002] Dashofy, E. M., van der Hoek, A. and Taylor, R. N.: "Towards architecture-based self-healing systems." in: Proceedings of the *Proceedings of the first workshop on Self-healing systems*. Charleston, South Carolina. Nov. 2002.

[De Meer and Koppen, 2005] De Meer, H. and Koppen, C.: "Self-Organization of Peer-to-Peer Systems", in: Steinmetz, R. and Wehrle, K. (eds.): *Peer-to-Peer Systems and Applications.* Springer (LNCS 3485). Berlin. 2005. (pp. 247-266).

[Dellarocas, 1998] Dellarocas, C.: "Toward Exception Handling Infrastructures for Component-Based Software", in: Proceedings of the *International Workshop on Component-based Software Engineering, in conjunction with the 20th International Conference of Software Engineering (ICSE)*. Kyoto, Japan. April 1998.

[DeRemer and Kron, 1976] DeRemer, F. and Kron, H.: "Programming-in-the-Large Versus Programming-in-the-Small", in: *IEEE Transactions on Software Engineering,* Vol. 2, No.2 (June 1976). June 1976. (pp. 321-327).

[Dornfest and Brickley, 2001] Dornfest, R. and Brickley, D.: "Metadata", in: Oram, A. (eds.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly. 2001. (pp. 191-202).

[Dourish and Bellotti, 1992] Dourish, P. and Bellotti, V.: "Awareness and Coordination in Shared Workspaces", in: Proceedings of the *Proceedings of the 4th ACM Conference on CSCW*. Toronto / Canada. 1992.

[Dustdar *et al.*, 2003] Dustdar, S., Gall, H. and Hauswirth, M.: *Software-Architekturen für Verteilte Systeme.* Springer, Berlin. 2003.

[Eberspächer and Schollmeier, 2005] Eberspächer, J. and Schollmeier, R.: "First and Second Generation of Peer-to-Peer Systems", in: Steinrücken, J. and Wehrle, K. (eds.): *Peer-to-Peer Systems and Applications.* Springer (LNCS 3485). Berlin. 2005.

[Eclipse, 2005] Eclipse: *The Eclipse Project*. 2005. (http://www.eclipse.org/)

[Ensel, 2001] Ensel, C., Keller, A.: "Managing Application Service Dependencies with XML and the Resource Description Framework." in: Proceedings of the *7th International IFIP/IEEE Symposium on Integrated Management (IM 2001)*. May 2001.

[Farnoudi, 2006] Farnoudi, A.: "Transaktionsmanagement in Service-Orientierten Peer-to-Peer Architekturen", in: Proceedings of the *Workshop Service-Oriented Architectures of the GI Jahrestagung*. GI Publishing. B-IT Bonn, Germany.

[Ferscha *et al.*, 2004] Ferscha, A., Hechinger, M., Mayrhofer, R. and Oberhauser, R.: "A Light-Weight Component Model for Peer-to-Peer Applications", in: Pro-

ceedings of the *24th International Conference on Distributed Computing Systems Workshop (ICDSW 2004)*. Tokio, Japan. March 2004. (pp. 520-527).

[Flanagan, 2005] Flanagan, D.: *Java in a nutshell - A desktop quick reference (5th edition for Java 5.0).* O'Reilly. 2005.

[Foster and Iamnitchi, 2003] Foster, I. and Iamnitchi, A.: "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid

Computing", in: Proceedings of the *2nd International Workshop on Peer-to-Peer Systems (IPTPS 03)*. 2003. (pp. 118-128).

[Frank, 1999] Frank, U.: "Component Ware - Software-technische Konzepte und Perspektiven für die Gestaltung betrieblicher Informationssysteme", in: *Information Management & Consulting,* Vol. 14, No.2. 1999. (pp. 11-18).

[Fraunhofer, 2005] Fraunhofer: *BSCW - Homepage*. 2005. (http://bscw.fit.fraunhofer.de/)

[Gamma *et al.*, 1995] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston. 1995.

[Garlan *et al.*, 2000] Garlan, D., Monroe, R. T. and Wile, D.: *ACME: Architectural Description of Component-Based Systems.* Technical Report, Carnegie Mellon University, Software Engineering Institute. 2000.

[Garlan and Shaw, 1993] Garlan, D. and Shaw, M.: "An introduction to software architecture", in: Ambriola, V. and Tortora, G. (eds.): *Advances in Software Engineering and Knowledge Engineering, volume 1.* World Scientific Publishing Company. 1993. (pp. 1-40).

[Gnasa *et al.*, 2005] Gnasa, M., Won, M. and Cremers, A. B.: "Three Pillars for Congenial Web Search - Continuous Evaluation for enhancing Web Search Effectiveness", in: *Journal of Web Engineering,* Vol. No. (pp. 252-280).

[Götz *et al.*, 2005] Götz, S., Rieche, S. and Wehrle, K.: "Selected DHT Algorithms", in: Steinmetz, R. and Wehrle, K. (eds.): *Peer-to-Peer Systems and Applications.* Springer (LNCS 3485). Berlin. 2005.

[Halepovic and Deters, 2002] Halepovic, E. and Deters, R.: "Building a P2P Forum System with JXTA", in: Proceedings of the *2nd IEEE International Conference on Peer-to-Peer Computing (P2P2002)*. Linköpings, Sweden. September, 2002.

[Hallenberger, 2000] Hallenberger, M.: *Programmierung einer interaktiven 3D-Schnittstelle am Beispiel einer Anpassungsschnittstelle für komponentenbasierte Anpassbarkeit.* Master Thesis, University of Bonn. Bonn, Germany. 2000.

[Hantschel *et al.*, 2006] Hantschel, R., Ruf, F. and Strotbek, H.: *Vergleich von BPEL Laufzeitumgebungen.* Fachstudie Nr. 54. Institut für Architektur von Anwendungssystemen. Stuttgart. 2006.

[Hasselmeyer, 2001] Hasselmeyer, P.: "Managing Dynamic Service Dependencies", in: Proceedings of the *12th International Workshop on Distributed Systems: Operations / Management (DSOM' 2001)*. Nancy, France. October 2001.

[Henderson and Kyng, 1991] Henderson, A. and Kyng, M.: "There's no place like home: Continuing design in use", in: Greenbaum, J. and Kyng, M. (eds.): *Design at Work.* Lawrence Erlbaum. Hillsdale. 1991.

[Hinken, 1999] Hinken, R.: *Verteilte komponentenbasierte Anpassbarkeit für Groupware - eine Laufzeit und Anpassungsumgebung.* Master Thesis, University of Bonn. Bonn, Germany. 1999.

[Hoeren, 2002] Hoeren, T.: "Urheberrecht und Peer-to-Peer-Dienste", in: Schoder, D., Fischbach, K. and Teischmann, U. (eds.): *Peer-to-Peer. Ökonomische, technologische und juristische Perspektiven.* Springer. Berlin. 2002.

[Holz *et al.*, 2002] Holz, K. P., Savidis, A., Schley, F. P. and Mejstrik, M.: *Berücksichtigung von Ausnahmefällen bei der kooperativen Bearbeitung von Projekten des konstruktiven Tiefbaus. Bericht über die erste Phase des DFG Projektes.* University of Cottbus and Technical University of Berlin. 2002.

[Hughes *et al.*, 1994] Hughes, J., King, V., Rodden, T. and Andersen, H.: "Moving out from the Control Room: Ethnography in System Design", in: Proceedings of the *CSCW'94*. Chapel Hill, ACM Press. (pp. 429-439).

[ICQ, 2005] ICQ: *ICQ Homepage*. 2005. (http://www.icq.com/)

[Intrinsyc, 2005] Intrinsyc: *Intrinsyc Software International, Inc. Homepage*. 2005. (http://j-integra.intrinsyc.com/)

[Jacobsen *et al.*, 1992] Jacobsen, I., Christerson, M., Jonsson, P. and Overgaard, G.: *Object-Oriented Engineering. A Use Case Driven Approach.* Addison-Wesley, Reading/Massachusetts. 1992.

[Kämpf, 2005] Kämpf, R.: *Virtuelle Organisation - zeitlich begrenzter Kooperations- und Leistungsverbund.* Report. Technischer Report des EBZ Beratungszentrum. http://www.ebz-beratungszentrum.de/organisation/themen/Virtuelle%20Unternehmen.html. 2005.

[Kan, 2001] Kan, G.: "Gnutella", in: Oram, A. (eds.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly and Associates, Inc. 2001.

[Katzmarzik, 2003] Katzmarzik, A.: *Entwicklung eines Softwareagenten zur Visualisierung und Aufbereitung von geometrischen und topologischen Daten aus ei-*

nem CAD-System. Diploma Thesis, Lehrstuhl für Ingenieurinformatik im Bauwesen, Uni Bochum. Bochum. 2003.

[Kaye, 2003] Kaye, D.: *Loosely Coupled: The missing pieces of Web Services.* RDS Press, Kentfield, USA. 2003.

[Keller and Kar, 2000] Keller, A. and Kar, G.: "Dynamic Dependencies in Application Service Management", in: Proceedings of the *International Conference on Parallel and Distributed Processing - Techniques and Applications.* Las Vegsa, USA. 2000.

[Kiczales *et al.*, 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: "Aspect-Oriented Programming", in: Proceedings of the *European Conference on Object-Oriented Programming (ECOOP)*. (pp. 220–242).

[Klamar, 2004] Klamar, S.: *Adaptive Architekturen für verteilte Systeme.* Master Thesis. University of Dresden. 2004.

[Kobayashi *et al.*, 1999] Kobayashi, N., Pierce, B. C. and Turner, D. N.: "Linearity and the Pi-Calculus", in: *ACM Transactions on Programming Languages and Systems,* Vol. 21, No.5. 1999. (pp. 914-947).

[Kon and Campbell, 2000] Kon, F. and Campbell, R. H.: "Dependence Management in Component-Based Distributed System", in: *IEEE Concurrency,* Vol. 8, No.1 (January-March). 2000.

[Kruchten, 2003] Kruchten, P.: *The Rational Unified Process. An Introduction.* Addison-Wesley Professional. 2003.

[Krüger, 2002] Krüger, M.: *Semantische Integritätsprüfung als Unterstützungsmechanismus für die Anpassung von komponentenbasierter Software.* Master Thesis, University of Bonn. Bonn, Germany. 2002.

[Leymann, 2001] Leymann, F.: *Web Services Flow Language (WSFL 1.0)*. IBM Software Group. 2001. (http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf)

[Liskov and Wing, 1993] Liskov, B. and Wing, J. M.: *Family Values: A Behavioral Notion of Subtyping.* MIT Report (MIT/LCS/TR-562b). MIT, USA. 1993.

[Lovelock and Vandermerwe, 1996] Lovelock, C. and Vandermerwe, S.: *Services Marketing.* Prentice-Hall, Inc. 1996.

[Lumpe, 1999] Lumpe, M.: *A pi-calculus Based Approach for Software Composition.* Inauguraldissertation der Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern. Bern, Schweiz. 1999.

[Magee *et al.*, 1995] Magee, J., Dulay, N., Eisenbach, S. and J., K.: "Specifying Distributed Software Architectures", in: Proceedings of the *5th European Software Engineering Conference*. Springer (LNCS 989). Barcelona, Spain. 1995.

[Malcom, 2005] Malcom, G.: "Component-based specification of distributed systems", in: Proceedings of the *2nd International Workshop on Formal Aspects of Component Software (FACS'05)*. Macao. October 2005.

[Marucci and Paternò, 2002] Marucci, L. and Paternò, F.: "Supporting Adaptivity with Heterogeneous Platforms through User Models", in: Proceedings of the *International Symposium on Mobile Human Computer Interaction*. Springer (LNCS 2411). Pisa, Italy. 2002.

[Marvie, 2002] Marvie, R., and Merle, P: *CORBA Component Model: Discussion and Use with OpenCCM.* Technical Report, Laboratorie de Recherche en Informatique de l'Université des Sciences et Technologies de Lille (LIFL). Lille, France. 2002.

[McCann and Huebscher, 2004] McCann, J. and Huebscher, M.: "Evaluation issues in Autonomic Computing", in: Proceedings of the *Grid and Cooperative Computing Workshops (GCC)*. Springer (LNCS 3252). (pp. 597–608).

[McIlroy, 1968] McIlroy, M. D.: "Mass Produced Software Components", in: (eds.): *Software Engineering, Report on a Conference sponsored by the NATO Science Committee.* Garmisch, Germany. 1968.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N.: "A Classification and Comparison Framework for Software Architecture Description Languages", in: *Journal of Software Engineering,* Vol. 26, No.1. 2000. (pp. 70-93).

[Milner, 1991] Milner, R.: *The polyadic pi-calculus: a Tutorial.* Technical Report ECS_LFCS_91_180, University of Edinburgh. Edinburgh Scotland. 1991.

[Milner, 1999] Milner, R.: *Communicating and mobile systems: the pi-calculus.* Cambridge University Bridge, Cambridge. 1999.

[Minar *et al.*, 2001] Minar, N., Hedlund, M. and Power, P.: "A Network of Peers. Peer-to-Peer Models through the History of the Internet", in: Oram, A. (eds.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly. 2001. (pp. 3-21).

[Mitrov, 2003] Mitrov, T.: *A Peer-to-Peer Environment for Component-Based, Tailorable Groupware Applications.* Master Thesis, University of Bonn. Bonn, Germany. 2003.

[Morch, 1997] Morch, A.: "Three Levels of End-User Tailoring: Customization, Integration, and Extension", in: Kyng, M. and Mathiassen, L. (eds.): *Computers and Design in Context.* The MIT Press. Cambridge, MA. 1997. (pp. 51-76).

[Mwaluseke and Bowen, 2001] Mwaluseke, G. W. and Bowen, J. P.: *UML Formalisation Literature Survey.* Technical Report, South Bank University (unpublished). London, England. http://myweb.lsbu.ac.uk/~mwalusgw/collections.pdf. 2001.

[Napster, 2005] Napster: *The (Commercial) Napster Homepage*. 2005. (http://www.napster.com/)

[Nielsen, 1993] Nielsen, J.: *Usability Engineering.* AP Professional, Boston. 1993.

[Nierstrasz and Achermann, 2003] Nierstrasz, O. and Achermann, F.: "A Calculus for Modelling Software Component", in: Proceedings of the *FMCO 2002*. Springer Verlag (LNCS 2852). 2003. (pp. 339-360).

[Oasis, 2002] Oasis: *UDDI Version 3.0.2, UDDI Spec Technical Committee Draft*. 2002. (http://uddi.org/pubs/uddi_v3.htm#_Toc12653717)

[OMG, 2002] OMG: *Corba Component Model 3.0 (CCM)*. 2002. (http://www.omg.org)

[Oram, 2001] Oram, A.: *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly. 2001.

[Oreizy *et al.*, 1999] Oreizy, P., Gorlick, M. M., Taylor, R. N. and Medividovic, N.: "An Architecture-based approach to Self-Adaptive Software", in: *IEEE Intelligent Systems, Trends and Controversies,* Vol. 12, No.May/June 1999.

[OSGI, 2004] OSGI: *About the OSGI Service Platform - Technical Whitepaper. Revision 3.0*. OSGI Alliance. 2004.

[Pahl, 2001] Pahl, C.: "A Pi-calculus based Framework for the Composition and Replacement of Components", in: Proceedings of the *OOPSLA'2001 - Workshop on Specification and Verification of Component-based Systems*. ACM Press. 2001.

[Palij, 2006] Palij, A.: *Selbstanpassbare Umgebung für P2P-Architekturen zur Behandlung von Laufzeitfehlern.* Master Thesis, Institut für Informatik III, University of Bonn. Bonn, Germany. 2006.

[Peltz, 2002] Peltz, C.: *Web Services Orchestration: a review of emerging technologies, tools, and standards.* Hewlett-Packard Company. 2002.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L.: "Foundations for the study of software architectures", in: *ACM SIGSOFT Software Engineering Notes,* Vol. 17, No.4. 1992. (pp. 40-52).

[Pierce and Sanigiorgi, 1993] Pierce, B. and Sanigiorgi, D.: "Typing and Subtyping for Mobile Processes", in: Proceedings of the *8th IEEE Logics in Computer Science*. Montreal, Canada. 1993.

[Pierce, 1997] Pierce, B. C.: "Foundational Calculi for Programming Languages." in: Tucker, A. B. (eds.): *The Computer Science and Engineering Handbook.* CRC Press. 1997. (pp. 2190-2207).

[Poizat *et al.*, 2004] Poizat, P., Royer, J. C. and Salaün, G.: "Formal Methods for Component Description, Coordination and Adaptation", in: Proceedings of the *WCAT'2004 - Int. Workshop on Coordination and Adaptation Techniques for Software Entities (ISBN : 84-688-6782-9)*. 2004. (pp. 89-100).

[Prieto-Diaz and Neighbors, 1986] Prieto-Diaz, R. and Neighbors, J.: "Module Interconnection Languages", in: *Journal of Systems and Software,* Vol. 6, No.4. 1986. (pp. 307-334).

[Radetzki and Cremers, 2004] Radetzki, U. and Cremers, A. B.: "IRIS: A Framework for Mediator-Based Composition of Service-Oriented Software", in: Proceedings of the *IEEE International Conference on Web Services (ICWS 2004)*. San Diego, California, USA. Juli 2004. (pp. 752-755).

[Rho *et al.*, 2006] Rho, T., Schmatz, M. and Cremers, A. B.: "Towards Context-Sensitive Service Aspects", in: Proceedings of the *Workshop on Object Technology for Ambient Intelligence and Pervasive Computing, in conjunction with 20th European Conference on Object Oriented Programming (ECOOP 06)*. Nantes, France. July 3-7, 2006.

[Rittenbruch *et al.*, 1998] Rittenbruch, M., Kahler, H. and Cremers, A. B.: "Supporting Cooperation in a Virtual Organization", in: Proceedings of the *International Conference on Information Systems (ICIS '98)*. Helsinki, Finnland. 1998. (pp. 30-38).

[Romanovsky, 2001] Romanovsky, A.: "Exception Handling in Component-Based System Development", in: Proceedings of the *International Computer Software and Application Conference, CompSAC*. IEEE Press. Illionois, USA. 2001. (pp. 580-586).

[Rüppel, 1999] Rüppel, U.: *Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau. Antrag auf Einrichtung eines DFG-Schwerpunktprogrammes.* Darmstadt, Germany. http://www.iib.bauing.tu-darmstadt.de/dfg-spp1103/en/definition/lang.html. 1999.

[Sametinger, 1997] Sametinger, J.: *Software Engineering with Reusable Components.* Springer Verlag, Berlin. 1997.

[Sangiorgi, 1993] Sangiorgi, D.: "From pi-calculus to Higher-Order pi-calculus --- and back." in: Proceedings of the *TAPSOFT'93*. Springer Publishing (LNCS 668). 1993. (pp. 151-166).

[Schneider, 2001] Schneider, J.: *Covergence of Peer and Web Services*. The O'Reilly Network. 2001. (http://www.oreillynet.com/pub/a/p2p/2001/07/20/convergence.html)

[Schoder and Fischbach, 2002] Schoder, D. and Fischbach, K.: "Peer-to-Peer - Anwendungsbereiche und Herausforderungen", in: Schoder, D., Fischbach, K. and Teischmann, U. (eds.): *Peer-to-Peer. Ökonomische, technologische und juristische Perspektiven.* Springer. Berlin. 2002.

[Scholl, 2005] Scholl, R.: *Napster Protocol Specification*. 2005. (http://opennap.sourceforge.net/napster.txt)

[Schwabe *et al.*, 2001] Schwabe, G., Streitz, N. A. and Unland, R.: *CSCW-Kompendium.* Springer Publishing, Berlin. 2001.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Inc. 1996.

[Shirky, 2001] Shirky, C.: "Listening to Napster", in: Oram, A. (eds.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly and Associates, Inc. 2001.

[Simon, 1991] Simon, S.: "Peer-to-Peer Network Management in an IBM SNA Network", in: *IEEE Network Magazine,* Vol. 5(?), No.March 1991. 1991. (pp. 30-34).

[Simons and Stafford, 2004] Simons, K. and Stafford, J.: "CMEH: Container Managed Exception Handling for Increased Assembly Robustnes", in: Proceedings of the *4th International Symposium on Component-based Software Engineering (CBSE7)*. Springer (LNCS 3054). Edinburgh, Scotland. (pp. 122-129).

[Slagter and Ter Hofte, 2002] Slagter, R. J. and Ter Hofte, G. H.: "End-user composition of groupware behaviour: The CoCoWare .NET architecture", in: Proceedings of the *ACM Conference on CSCW 2002*. ACM Press, New York. 2002.

[Sommerville, 2004] Sommerville, I.: *Software Engineering, 7th edition.* Addison Wesley. 2004.

[Steinmetz and Wehrle, 2006] Steinmetz, R. and Wehrle, K.: "What is This "Peer-to-Peer" About?" in: Steinmetz, R. and Wehrle, K. (eds.): *Peer-to-Peer Systems and Applications.* Springer Publishing (LNCS 3485). 2006.

[Stephanidis, 2001] Stephanidis, C.: "Adaptive Techniques for Universal Access", in: *Journal of User Modeling and User-Adapted Interaction (UMUAI),* Vol. 11, No. 2001. (pp. 159-179).

[Stiemerling, 2000] Stiemerling, O.: *Component-Based Tailorability.* Dissertation, Institut für Informatik III, University of Bonn. Bonn, Germany. 2000.

[Stiemerling *et al.*, 1999] Stiemerling, O., Hinken, R. and Cremers, A. B.: "The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware." in: Proceedings of the *IEEE Enterprise Computing Conference (EDOC 1999)*. Mannheim, Germany. 1999.

[Sun, 2000] Sun: *Java Beans Specification, V1.01*. 2000. (http://java.sun.com/products/java-beans/docs/spec.html)

[Sun, 2001] Sun: *Project JXTA: An Open, Innovative Collaboration*. Technical Report of Sun Microsystems. 2001. (http://www.jxta.org/project/www/docs/OpenInnovative.pdf)

[Sun, 2002] Sun: *Enterprise Java Beans Specifications V1.1 and V2.0.* 2002. (http://java.sun.com/products/ejb/docs.html/)

[Sun, 2005a] Sun: *JXTA v2.0 Protocols Specification (Revision 2.3.5)*. 2005a. (http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html)

[Sun, 2005b] Sun: *Project JXTA: Java Programmer's Guide*. 2005b. (http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf)

[Szyperski *et al.*, 2002] Szyperski, C., Gruntz, D. and Murer, S.: *Component Software - Beyond Object-Oriented Programming. .* Addison-Wesley, London. 2002.

[Taylor, 1996] Taylor, R. N.: "A Component- and Message-Based Architectural Style for GUI Software", in: *IEEE Transactions on Software Engineering,* Vol. 22, No.6.

[Thatte, 2001] Thatte, S.: *XLANG - Web Services for Business Process Design*. Microsoft. 2001. (http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm)

[Theiss *et al.*, 2004] Theiss, M., Meissner, U. and Rüppel, U.: "Network-Based Fire Engineering Supported by Agents", in: Proceedings of the *10th International Conference on Computing in Civil and Building Engineering (ICCCBE-X)*. Weimar, Germany.

[Tiwana, 2003] Tiwana, A.: "Affinity to Infinity in Peer-to-Peer Knowledge Platforms", in: *Communications of the ACM,* Vol. 46, No.5. (pp. 76-80).

[Tolksdorf, 2003] Tolksdorf, R.: "A Dependency Markup Language for Web Services", in: Chaudhri, A. B., Jeckle, M., Rahm, E. and Unland, R. (eds.): *Web, Web-Services, and Database Systems. NODe 2002 Web and Database-Related Workshops (Proceeding).* Springer, LNCS 2593. 2003. (pp. 129-140).

[Turner and Budgen, 2003] Turner, M. and Budgen, D.: "Turning Software into a service", in: *IEEE Computer,* Vol. 36, No.10. 2003. (pp. 38-45).

[Varki and Wong, 2003] Varki, S. and Wong, S.: "Consumer Involvement in Relationship Marketing of Services", in: *Journal of Service Research,* Vol. 6, No.1. (pp. 83-91).

[Verma *et al.*, 2004] Verma, K., Akkiraju, R., Goodwin, R., Doshi, P. and Lee, J.: "On Accommodating Inter Service Dependencies in Web Process Flow Composi-

tion", in: Proceedings of the *AAAI Spring Symposium on Semantic Web Services*. March, 2004. (pp. 37-43).

[W3C, 2001] W3C: *Web Services Description Language (WSDL) v1.1*. 2001. (http://www.w3c.org/TR/wsdl)

[W3C, 2002] W3C: *Web Service Choreography Interface (WSCI) 1.0 W3C Note 8 August 2002*. 2002. (http://www.w3.org/TR/wsci/)

[W3C, 2004a] W3C: *Extensible Markup Language (XML)*. 2004a. (http://www.w3.org/XML/)

[W3C, 2004b] W3C: *Simple Object Access Protocol (SOAP) v1.2*. 2004b. (http://www.w3c.org/TR/SOAP)

[W3C, 2004c] W3C: *Web Services Choreography Description Language Version 1.0 (WS-CDL). Working Draft*. 2004c. (http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/)

[W3C, 2004d] W3C: *Web Services Choreography Description Language Version 1.0 W3C Working Draft 17.* http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/. 2004d.

[W3C, 2004e] W3C: *Web Services Glossary, W3C Working Group Note 11 February 2004*. 2004e. (http://www.w3.org/TR/ws-gloss/)

[Wang and Vassileva, 2003] Wang, Y. and Vassileva, J.: "Trust and Reputation Model in Peer-to-Peer networks", in: Proceedings of the *3rd IEEE Peer-to-Peer Conference (P2P2003)*. IEEE Press. Linköping, Sweden. 2003.

[Wojciechowski and Weinhardt, 2002] Wojciechowski, R. and Weinhardt, C.: "Web Services und Peer-to-Peer Netzwerke", in: Schoder, D., Fischbach, K. and Teischmann, U. (eds.): *Peer-to-Peer. Ökonomische, technologische und juristische Perspektiven.* Springer. Berlin. 2002.

[Won, 2004] Won, M.: *Interaktive Integritätsprüfung für komponentenbasierte Architekturen. Technische Unterstützung für Endanwender beim Anpassen komponentenbasierter Software.* Dissertation, Institut für Informatik III, University of Bonn. Bonn, Germany. 2004.

[Won and Cremers, 2002] Won, M. and Cremers, A. B.: "Supporting End-User Tailoring of Component-Based Software - Checking Integrity of Composition", in: Proceedings of the *Proceedings of Colognet 2002 (Conjuction with LOPSTR 2002)*. Madrid, Spain. 2002.

[Wulf, 1999] Wulf, V.: "Let's see your Search-Tool! - Collaborative use of Tailored Artifacts in Groupware." in: Proceedings of the *GROUP '99*. Phoenix, USA. (pp. 50-60).

[Wulf *et al.*, 2006] Wulf, V., Pipek, V. and Won, M.: "Component-based Tailorability: Towards highly flexible software applications", in: *Int. Journal on Human-Computer Studies,* Vol. accepted for publication, No. 2006.

[Yau, 2001] Yau, C.: *Creating Peer-to-Peer Middleware from Web Services Technologies*. Technical Report, Intel Co. 2001. (http://www.intel.com/technology/magazine/computing/it09016.pdf)

[Zadeh, 1963] Zadeh, L. A.: "On the definition of adaptivity", in: Proceedings of the *IEEE (Correspondence)*. March 1963. (pp. 469).

[Zaremski and Wing, 1997] Zaremski, A. M. and Wing, J. M.: "Specification Matching of Software Components", in: *ACM Transactions on Software Engineering and Methodology,* Vol. 6, No.4. 1997. (pp. 333-369).

Appendix A: The pi-calculus

Syntactical Elements – Names, Actions and Processes

The most primitive element in the pi-calculus is a *name*. Names have two intentions, namely that of a (communication) *channel* and that of *messages*. Names correspond to an infinite set $a, b, c, , \ldots \in \Pi$ without any structure. Names are the base for building *actions*. An action π represents either sending or receiving a message (a name), or making a silent transition. The syntax is:

$$\begin{aligned} \pi ::= \; & x(y) && \textit{receive } y \textit{ along } x \\ & \bar{x}\langle y \rangle && \textit{send } y \textit{ along } x \\ & \tau && \textit{unobservable action} \end{aligned}$$

Actions as declared above constitute the fundament for building *processes*. A process can be produced by the following syntax [Milner, 1999]:

$$P \; ::= \; \sum_{i \in I} \pi_i.P_i \;\Big|\; P_1 \mid P_2 \;\Big|\; \nu\, a\; P \;\Big|\; !P \;\Big|\; 0$$

where I is any finite indexing set. The processes $\sum_{i \in I} \pi_i.P_i$ are termed summation. With that construct, *sequential process behavior* can be formalized: action π_i must occur before process P_i becomes active.

The process construct $P_1 \mid P_2$ indicates the *concurrency* of processes. Here, the processes P_1 *and* P_2 run concurrently. The restriction operator $\nu\, a\; P$ introduces a local and fresh channel a in P. That is, messages sent and received by P on x are never mixed with messages sent and received on any other channel generated elsewhere, even another channel that happens to be named a. The construct $!P$ indicates the *replication* of a process P. This means that there are infinitely many processes P concurrently active. Formally, the replication operator can be formalized as $!P \rightarrow P \mid !P$. The nil process, written 0, is a process whose execution is complete and has stopped.

The pi-calculus is a process *algebra*, that is, (more) complex constructs can be composed from the existing small expressions. This way, the calculus can intuitively be extended by higher-level constructs. It is feasible to encode each expression following the lambda-calculus by an adequate pi-calculus expression (see prove in [Pierce, 1997], section 3.2).

Operational Semantics

The *operational semantics* of the pi-calculus are defined as a *reduction relation* $\rightarrow$ over processes. $P \rightarrow Q$ means that P can be transformed into Q by a single computa-

tional step. The basic reduction rule captures the ability of processes to communicate through channels:

$$COMM : \bar{x}y.P \mid x(z).Q \rightarrow P \mid [y/z]Q$$

Communication between two processes can only take place, if the channel names of their action prefixes are equal. In addition, both channels must be *complementary,* that is, an input and an output prefix can react only. $[y/z]Q$ denotes a process Q in which the name y has been substituted for the name z. COMM is actually the only *axiom* for the reduction relation $\rightarrow$. There are two other *inference rules* applicable, indicating that reduction can occur underneath composition and restriction.

$$COMPOSITION : \frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R}$$

$$RESTRICTION : \frac{P \rightarrow Q}{(v\,x)P \rightarrow (v\,x)Q}$$

Furthermore, a set of equations can be used to rearrange a given process expression and, thus, to make reduction rules more easy. Two process expressions P and Q are *structurally congruent* (written $P \equiv Q$), if one can transform one into the other by using the following equations :

Structural Congruence :

$P \mid Q \equiv Q \mid P$ (*communitativity of parallel composition*)

$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ *(associativity of parallel composition*

$(v\,a\,)P \mid Q \equiv (v\,a)(P \mid Q)$ *(scope extrusion)*

$!\,P \equiv P \mid !P$ *(replication)*

A more precise explanation on structural congruence can be found in [Milner, 1991].

Extensions to the Original pi-Calculus

A couple of extensions and variants have been proposed for enriching the syntax of the conventional pi-calculus. A complete survey of these variants would certainly go beyond the scope of this work. Instead, only those variants are introduced briefly that this work uses for formalizing the $SO_{P2P}A$ architectural style. Note that all these extensions explained in the following are appreciated variants of the conventional pi-calculus. Further related notions used for the formalization of $SO_{P2P}A$ are compared in the related work section of chapter 4 (see section 4.4).

The so-called *polyadic pi-calculus* accomplishes the communication of a tuple of names aggregated in terms of a single message on a channel. The input and output prefixes for communicating these tuples are defined as follows:

$$output\ prefix : \bar{x}\langle y_1, \ldots, y_n \rangle.P$$

$$input\ prefix : x(z_1, \ldots, z_n).Q$$

These prefixes can be encoded by the conventional (also called *monadic*) pi-calculus by passing the name of a private channel through which the multiple arguments are then passed consecutively (see more information of this encoding in [Pierce, 1997]):

$$\bar{x}\langle y_1, \ldots, y_n \rangle.P = (v\,p)\bar{x}p.\bar{p}y_1.\ldots.\bar{p}y_n.P$$

$$x(z_1, \ldots, z_n).Q = x(p).p(z_1).\ldots.p(z_n).Q$$

A *sum* operator + is often added to the syntax in order to model the nondeterministic choice between two sub-processes P and Q: $(P + Q)$. In other words, the process $R = (P + Q)$ behaves either like P or like Q. The sum operator is beneficial for formalizing systems with potentially many parallel processes, whereas not all processes are used or invoked at the same time.

The *higher order pi-calculus* ($\mathrm{HO}\pi$) allows to send and to receive processes instead of names through channels. The input and output prefixes are adapted appropriately to distinguish between names and processes as input and output values:

$$\textit{output prefix} : \bar{x}[A].P$$

$$\textit{input prefix} : x[B].Q$$

The following reduction rule defines the operational semantics for passing processes along channels:

$$MIGRATE : \bar{x}[A].P \mid x[B].Q \rightarrow P \mid [A/B]Q$$

After applying the reduction rule, process A has been migrated to the process Q. The syntax, the semantics, and the rationale of the higher-order pi-calculus have been studied by Sangiorgi [Sangiorgi, 1993]. Sangiorgi proved the equivalence of higher-order and monadic pi-calculus, that is, the ability to pass processes does not increase the expressivity of the pi-calculus at all. A simulation can be carried out by transmitting a name that acts a pointer to a process P.

Lebenslauf

Name:	Sascha Josef Alda
Geburtstag:	1974
Geburtsort:	Königswinter
Nationalität:	Deutsch
Schulausbildung:	1981-1985: Katholische Grundschule Eudenbach 1985-1991: Realschule Oberpleis 1991-1994: Gymnasium am Oelberg, Abschluss: Abitur
Studium:	10/1994 – 9/2000: Studium der Informatik mit Schwerpunkt Wirtschaftsinformatik an der Universität Koblenz-Landau, Abteilung Koblenz 4/98 – 8/98: Studium der Informatik an der Rijksuniversiteit Leiden (Niederlande), als Stipendiat des Erasmus Programms der EU 9/2000: Abschluss des Studiums. Akademischer Grad: Diplom Informatiker (Dipl.-Inform.)
Berufspraktische Erfahrungen (Auszug):	
	8/1996 - 3/1998: Studentische Hilfskraft an der Wissenschaftlichen Hochschule der Unternehmensführung (WHU), Lehrstuhl für Marketing, Univ.-Prof. Dr. habil Dr. h.c. Christian Homburg**.** 9/1998 – 8/2000: Studentische Hilfskraft am Institut für Software Technologie (IST) an der Universität Koblenz-Landau, Univ.-Prof. Dr. habil. Jürgen Ebert.
Beruflicher Werdegang:	10/2000 – 6/2001: Software Entwickler bei der IBM Deutschland, Abt. Forschung und Entwicklung in Böblingen. seit 6/2001: Wissenschaftlicher Mitarbeiter und Doktorand am Institut für Informatik III der Rheinischen Fried-

rich-Wilhelms-Universität zu Bonn, Univ.-Prof. Dr. habil. Armin B. Cremers.

Publikationen: 6/2001 – heute: über 20 Publikationen in Journals, Tagungsbänder von Konferenzen und Workshops, Tätigkeitsberichte (u.a. für die DFG).