

Modeling Telecommunication Systems: From Standards to System Architectures

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des
akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte
Dissertation

vorgelegt von

Dipl.-Ing., Dipl.-Wirt. Ing.
Dominikus Herzberg
aus Bonn

Berichter: Univ.-Prof. Dr.-Ing. Manfred Nagl
Univ.-Prof. Dr. rer. nat. Manfred Broy, TU München

Tag der mündlichen Prüfung: 17. September 2003

Diese Dissertation ist auf den Internetseiten der
Hochschulbibliothek online verfügbar.

Abstract

The architecture of a technical system reflects significant design decisions about the system's organization and typically documents a description of key system elements (be they hardware or software), their composition, functioning, and interrelation. The process of creating a description of an architecture is called *architecture modeling*. In the telecommunication domain, the architecture level has always played an important role in the design and evolution of communication systems and networks.

However, the way how telecommunication engineers describe their architectures is surprisingly rudimentary: They use natural languages and conceptual drawings, as a look into "old" as well as recent standards unveils. Even in the transition phase from standards to the early design phases of systems development, system designers do not go much beyond that *level of informality*. Therefore, as practice shows, in telecommunications, architecture modeling but not the understanding of architecture as such lacks (i) a suitable, consistent and formal modeling language, which is adapted to the needs of systems designers, and (ii) a methodology to support the modeling process. This work addresses these deficiencies.

In this thesis, a systematic approach is presented for modeling architectures of virtually any telecommunication system. This includes a *methodology*, a *modeling language*, and a prototype *implementation* of the language. A major contribution of this work is the statement that such an approach can be based upon as few as three basic cornerstones for a networked system: the types of communication and the design principles of distribution and layering. The investigation distills fundamental insights for the design and construction of modern communication systems.

The outcome can be summarized as follows: The aspect of control leads to the distinction of three elementary *types of communication* (control-oriented, data-oriented, and protocol-oriented communication) and provides the rationale for grey-box architecture descriptions. The aspect of distribution can be manifested by the notion of a *complex connector*, which is the key concept to model connection-oriented, connectionless and even space-based communication networks including quality of service. Layering in telecommunication systems is different from

the ordinary understanding of the term. Layers in a distributed communication system follow a generic form of refinement, namely *communication refinement*. Communication refinement constitutes a true abstraction hierarchy, which can be interpreted from two perspectives: from a node-centric and from a network-centric viewpoint. The viewpoint chosen has an important impact on the systems understanding.

The foundation of this work is mathematical, its application is practical. The mathematics help giving precise definitions on the notions of distribution and layering; the resulting implications shape the methodology and the language. The language developed is based on ROOM (Real-Time Object-Oriented Modeling), an object-oriented but also component-based language. Key language features of ROOM will be integrated in the forthcoming 2.0 release of the Unified Modeling Language, UML. The *extensions* proposed to *ROOM* led to a careful redesign of the language and a prototype implementation. The accompanying methodology is organized in *method blocks*, each block being a self-contained methodological unit encompassing heuristics and architectural solution patterns.

The thesis statement is supported by a real-life *case study* on the SIGTRAN (SIGnaling TRANsport) architecture. In the case study, first the understanding of architecture models as imposed by standards is presented. At the end of this work, it is shown that systematic architecture modeling is relatively easy and comes at little cost – the gains in terms of clarity, preciseness and expressiveness are remarkable.

Zusammenfassung

Die Architektur eines technischen Systems spiegelt bedeutsame Entwurfsentscheidungen über den Systemaufbau wider. Sie dokumentiert eine Beschreibung der Schlüsselemente des Systems (Hardware und/oder Software), ihre Komposition, Funktionsweise und wechselseitigen Bezüge. Der Vorgang der Erstellung einer Architekturbeschreibung wird auch *Architekturmodellierung* genannt. In der Telekommunikation hat die Architekturebene stets eine wichtige Rolle im Entwurf und in der Weiterentwicklung von Kommunikationssystemen und -netzen gespielt.

Jedoch ist die Art und Weise, wie Telekommunikationsingenieure ihre Architekturen beschreiben, erstaunlich rudimentär: Sie nutzen natürliche Sprache und Konzeptdiagramme, wie ein Blick in ältere als auch jüngere Standards offenbart. Selbst im Übergang von den Standards hin zu den frühen Entwurfsphasen in der Systementwicklung gehen die Systementwickler kaum über diesen *informellen Grad* der Darstellung hinaus. Wie sich in der Praxis zeigt, mangelt es bei der Architekturmodellierung – nicht jedoch bei dem Verständnis von Architektur an sich – (i) an einer geeigneten, konsistenten und formalen Modellierungssprache, die an die Bedürfnisse der Systementwickler angepasst ist und (ii) an einer Methodik, die den Modellierungsprozess unterstützt. Mit diesen Mängeln befasst sich die vorliegende Arbeit.

In dieser Doktorarbeit wird ein systematischer Ansatz zur Architekturmodellierung von praktisch beliebigen Telekommunikationssystemen vorgestellt. Dies beinhaltet eine *Methodik*, eine *Modellierungssprache* und eine prototypische *Implementierung* dieser Sprache. Ein wesentlicher Beitrag dieser Arbeit ist die Behauptung, dass ein solcher Ansatz auf nur drei grundlegenden Eckpfeilern basiert: den Arten von Kommunikation und den Entwurfsprinzipien von Verteilung und Schichtung. Die Untersuchung liefert grundlegende Einsichten zum Design und Aufbau moderner Kommunikationssysteme.

Das Ergebnis kann wie folgt zusammengefasst werden: Der Aspekt der Kontrolle führt zur Unterscheidung von drei elementaren *Kommunikationsarten* (kontroll-, daten- und protokoll-orientiert) und begründet „grey-box“ Architekturbeschreibungen. Der Gesichtspunkt der Verteilung manifestiert sich im Begriff des

komplexen Konnektors, der sich als Schlüsselkonzept zur Modellierung von verbindungslosen, verbindungsorientierten und sogar von „space-based“ Kommunikationsnetzen inklusive ihrer Dienstqualitäten herausprägt. Der Begriff der Schichtung hat in Telekommunikationssystemen eine andere Bedeutung als sie sonst üblich ist. In einem verteilten System folgen die Schichten einer generischen Form der Verfeinerung, der *Kommunikationsverfeinerung*. Die Kommunikationsverfeinerung begründet eine echte Abstraktionshierarchie, die aus zwei Perspektiven interpretiert werden kann: aus einer knoten- und aus einer netzwerk-zentrischen Sicht. Die gewählte Sichtweise hat einen entscheidenden Einfluss auf das Systemverständnis.

Diese Arbeit ist einerseits mathematisch fundiert, andererseits ist sie anwendungsorientiert. Mit Hilfe der Mathematik werden präzise Definitionen des Verständnisses von Verteilung und Schichtung gegeben; die sich ergebenden Implikationen prägen Methodik und Sprache. Die entwickelte Sprache basiert auf ROOM (Real-Time Object-Oriented Modeling), einer objekt-orientierten aber auch komponenten-orientierten Sprache. Wichtige Eigenschaften von ROOM werden in der bevorstehenden Version 2.0 der Unified Modeling Language, UML, integriert sein. Die vorgeschlagenen *ROOM-Erweiterungen* führen zu einer Neugestaltung der Sprache und einer Prototyp-Implementierung. Die begleitende Methodik ist in *Methodenblöcken* organisiert, wobei jeder Block eine abgeschlossene Einheit bildet und Heuristiken und Architektur-Lösungsmuster enthält.

Die Aussagen dieser Arbeit werden durch eine Fallstudie aus der Praxis gestützt. In der Fallstudie zur SIGTRAN (SIGnaling TRANsport) Architektur wird zunächst dargestellt, welches Verständnis über Architekturmodelle den Standards zugrunde liegt. Es zeigt sich gegen Ende dieser Arbeit, dass ein systematischer Ansatz zur Architekturmodellierung relativ einfach und mit nur wenig Kosten verbunden ist – die Zugewinne im Sinne von Klarheit, Genauigkeit und Ausdrucksstärke sind jedoch bemerkenswert.

Contents

1	Introduction	1
1.1	Problem Description	2
1.2	Thesis Statement and Approach	5
1.2.1	Thesis Statement	5
1.2.2	Approach	7
1.2.3	A Note to the Reader	8
1.3	Scientific Contribution	9
1.4	Case Study: SIGTRAN	11
1.4.1	Background	11
1.4.2	The SIGTRAN Architecture	11
1.4.3	Problems and Curiosities	16
1.5	Overview	22
2	Foundations	29
2.1	Reference Models: OSI and TCP/IP	30
2.1.1	Historical Background	30
2.1.2	The Architecture of OSI RM	31
2.1.3	The Architecture of TCP/IP RM	36
2.1.4	Conclusions	40
2.2	From (Embedded) Real-Time Systems to Communication Networks	43
2.2.1	(Embedded) Real-Time Systems	43
2.2.2	An Extended Model of Real-Time Systems	45
2.2.3	Collaborating Distributed Real-Time Systems	49
2.2.4	Collaborating Networks	51
2.2.5	Summary	51
2.3	A Brief Primer on ROOM	53
2.3.1	Structural Elements	53
2.3.2	Behavioral Elements	59
2.3.3	Model Execution	60
2.3.4	ROOM Tools	61
2.4	Mathematical Formalism	64

2.4.1	The Concepts of Streams, Channels, and Components . . .	64
2.4.2	Composition with Mutual Feedback and Behavioral Re- finement	67
2.5	Summary	69
3	Types of Communication	71
3.1	Problems with the Client-Server and the Peer-to-Peer Communi- cation Model	72
3.2	Alignments of Control	74
3.2.1	What is Control?	74
3.2.2	Who controls whom?	76
3.2.3	One-Sided Control-Oriented Communication	78
3.2.4	Two-Sided Control-Oriented Communication	84
3.2.5	Zero-Sided Control-Oriented Communication	87
3.2.6	Annotating Communication Types in ROOM	87
3.3	Communication Services in Telecommunication Systems	90
3.3.1	Connection-Oriented Communication exemplified on TCP	91
3.3.2	Connectionless Communication exemplified on UDP . . .	101
3.4	Resource Control in Telecommunication Systems	105
3.4.1	Resource Control exemplified on MGCP	106
3.5	Summary	110
4	Distribution	115
4.1	What is Distribution?	116
4.1.1	Definitions in Literature	116
4.1.2	An Algebraic Model of Distribution	117
4.1.3	The Complex Connector	119
4.2	Introducing the Complex Connector in ROOM	121
4.2.1	Discussion of Solutions	121
4.2.2	The Extension: Typed Bindings	121
4.2.3	The Algebraic Definition Visualized	127
4.3	Networked Communication	128
4.3.1	Specialty of the Approach	128
4.3.2	Connection-oriented Communication Networks	130
4.3.3	Connectionless Communication Networks	135
4.3.4	Space-Based Communication Networks	143
4.4	Addressing	145
4.4.1	Introducing Addressing Concepts in ROOM	145
4.4.2	Modeling Address Hierarchies	150
4.5	Summary	155

5	Layering	159
5.1	What is Layering?	160
5.1.1	Definitions in Literature	160
5.1.2	An Algebraic Model of Layering	161
5.1.3	The Abstraction Hierarchy and Implications on Architec- ture Design	165
5.2	Layering in ROOM	170
5.2.1	Criticism on ROOM's Interlayer Model	170
5.2.2	Improvements to ROOM's Interlayer Model	173
5.2.3	Layering versus Layering	178
5.2.4	Node-Centric and Network-Centric Designs in ROOM	180
5.3	Layered Communication Networks	184
5.3.1	Layered Networks exemplified on TCP	184
5.3.2	Layered Networks exemplified on UDP	185
5.3.3	Layered Networks exemplified on MGCP	186
5.4	Planes in Communication Networks	190
5.4.1	The Plane Concept in Telecommunications	190
5.4.2	Introducing Planes in ROOM	191
5.4.3	Architecture Modeling with Planes	193
5.5	Summary	196
6	Language and Implementation	199
6.1	The Design of the ROOM Language	200
6.1.1	The Four Layer Meta-data Architecture	200
6.1.2	From a (Semi)Formal Specification towards a Meta-Model	201
6.1.3	ROOM Meta-Model	204
6.2	The Design of ROOM++	206
6.2.1	Summary of Enhancements to ROOM	206
6.2.2	ROOM++ Meta-Model	213
6.3	The Implementation: PyROOM++	219
6.3.1	Features and Accepted Shortcomings	220
6.3.2	Implementation of Four Layer Meta-data Architecture	222
6.3.3	Description of Basic Functioning	223
6.4	Improvement to High-Level Behavior Specification	226
6.4.1	Problem Description	226
6.4.2	The Concept of Coupled State Machines	228
6.4.3	Extending the UML	230
6.4.4	Conclusion	232
6.5	Summary	234

7	Methodology	237
7.1	Systematic Approach: From Standards to System Architectures . . .	238
7.1.1	Method Block: System Network Architecture	239
7.1.2	Method Block: Protocol Entity	245
7.1.3	Method Block: Resource Entity	248
7.1.4	Method Block: Aspect Entity	250
7.2	The Case Study Revisited	252
7.2.1	Overview	253
7.2.2	Step 1: MTP3-User Communication Network	254
7.2.3	Step 2: MTP3/M3UA Communication Network	256
7.2.4	Step 3: MTP3 Communication Service	257
7.2.5	Step 4: M3UA Communication Service	258
7.2.6	Step 5: Mediator	260
7.2.7	Step 6: Design View	261
7.2.8	Step 7: Media Gateway	262
7.2.9	Evaluation	263
7.3	Experiences	265
7.4	Summary	267
8	Related Work	269
8.1	Historical Context	270
8.2	Position of ROOM to Related Languages	273
8.3	Modeling Telecommunication Systems	277
8.4	Frameworks	279
8.5	Architecture	280
9	Summary and Outlook	283
9.1	Summary	284
9.1.1	Cornerstones	284
9.1.2	Results	285
9.2	Outlook	287

List of Figures

1.1	The ISDN Reference Model	12
1.2	SIGTRAN Functional Model	13
1.3	SIGTRAN Signaling Transport Components and Protocol Architecture	15
1.4	SIGTRAN with M3UA: ISUP Message Transport	16
2.1	OSI seven layer reference model, see [ITU94, p.31]	31
2.2	Layers, SAPs and CEPs according to OSI	32
2.3	Finite state machine of the SAP	36
2.4	The TCP/IP reference model, taken from [Tan96, p.36].	37
2.5	A simple server using TCP services written in Python	39
2.6	A simple client using TCP services written in Python	40
2.7	The basic elements of a real-time system	44
2.8	An extended model for real-time systems	46
2.9	Collaborating distributed real-time systems	50
2.10	Collaborating networks	51
2.11	Actor class with ports	54
2.12	Actor class containing all types of actor references	56
2.13	Layer connection, see also [SGW94, p.202]	58
2.14	Export connection, see also [SGW94, p.205]	58
2.15	Relation of the application, the ROOM VM and the target environment, see also [SGW94, p.325]	61
2.16	Specification S with the syntactic interface $(I \triangleright O)$	66
3.1	A simple controller/controllee scenario	74
3.2	State model of the train resource	75
3.3	Sequence diagrams of the message exchange between controller and controllee	77
3.4	One-sided control-oriented communication	79
3.5	State model of the controller	80
3.6	Domain model of the controller, component-oriented	80

3.7	Structural pattern for actor that exerts control	82
3.8	Grey-box specification of an actor	82
3.9	Domain model of the controller, object-oriented	83
3.10	Two-sided control-oriented communication	85
3.11	Communication types: (a) control-oriented, (b) protocol-oriented, (c) data-oriented	88
3.12	Example of control-oriented communication via a relay actor . . .	89
3.13	Connection-oriented and connectionless communication services exemplified on TCP and UDP	90
3.14	Actor model of a TCP service user, object-oriented	94
3.15	Model of the relation TCP service user/service provider	96
3.16	Format of a TCP segment, see [Pos81b]	97
3.17	The TCP FSM figure is derived from [Tan96, p.532]. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. User commands are given in bold font.	98
3.18	Model of a connection-oriented communication service	100
3.19	Format of a UDP datagram, see [Pos80]	103
3.20	Model of a connectionless communication service	103
3.21	Model of the Media Gateway Controller	107
3.22	Model of a resource control relation	108
3.23	Summary of communication types: (a) control-oriented, (b) protocol- oriented, and (c) data-oriented	110
3.24	Vertical communication types for (a) connection-oriented, (b) con- nectionless, and (c) resource services	111
3.25	Horizontal communication types for (a) a protocol-oriented and (b) a data-oriented protocol	112
4.1	Modeling Quality of Service (QoS) attributes	120
4.2	Options to introduce complex connectors in ROOM: (a) actor class, (b) enhanced binding concept	122
4.3	Notation for typed binding: (a) bidirectional, (b) unidirectional; binding name is optional	123
4.4	Example: En-/decoding complex connectors	124
4.5	Extended notation for typed bindings: (a) fixed, (b) optional, (c) im- ported	125
4.6	Example of an invalid specification with typed bindings	126
4.7	Equation 4.1 and 4.2 visualized: (a) functional breakup, (b) distri- bution	127
4.8	Modeling Distribution: Communication network of entities com- municating via a communication service	129

4.9	TCP and UDP from the viewpoint of networked communication of independent layers (compare to figure 3.13)	130
4.10	Model of the TCP user layer as an independent, abstract network .	132
4.11	TCP user network, run-time scenario	133
4.12	Address structure characterizing connection-oriented communication	136
4.13	Address structure characterizing connectionless communication .	137
4.14	Model of the UDP user layer as an independent, abstract network .	138
4.15	UDP user network, run-time scenario	141
4.16	Model of the (a) TCP and (b) UDP provider layer as an independent, abstract network	142
4.17	JavaSpaces user network model	144
4.18	The notation for addresses and address associations exemplified .	147
4.19	Elaborated model for TCP user network	151
4.20	Possible refinement for TCPUDeMux	152
4.21	Elaborated model for UDP user network	153
4.22	Combined model for the TCP/UDP provider network	154
4.23	Notation for typed binding	155
4.24	Generalization of communication services: (a) connection-oriented (CON), (b) connectionless (CNL)	156
4.25	Example for modeling addresses, address spaces and address association	157
5.1	A schematic model of layering according to OSI RM and TCP/IP RM	162
5.2	Mathematical model of layered distributed communication	163
5.3	Layered model: the abstraction hierarchy	166
5.4	Node-centric design approach	168
5.5	Network-centric design approach	169
5.6	Layers in ROOM, see [SGW94, p.201]	170
5.7	Coordinator capsule bridging layers, see [SGW94, p.208]	172
5.8	Notation for SPPs, SUPs and layer contracts	174
5.9	Explicit export notation for SPPs and SUPs	175
5.10	Communication types between layers: (a) control-oriented, (b) protocol-oriented, (c) data-oriented	176
5.11	Patterns of vertical communication in a communication system: (a) connection-oriented, (b) connectionless, and (c) resource control services	177
5.12	The semantics of layering exemplified	178
5.13	Node-centric communication refinement in ROOM++	181
5.14	Network-centric communication refinement in ROOM++	182

5.15	The TCP user network and the TCP provider network set into relation	184
5.16	The UDP user network and the UDP provider network set into relation	186
5.17	Model of a distributed resource control relation	187
5.18	Refinement of complex connector between MGC and MG	188
5.19	The MGC node	189
5.20	Informal model of the ISDN/GSM system architecture; taken from [EV98, p.117]	190
5.21	Planes exemplified	192
5.22	Precise architectural model based on ISDN, fulfilling figure 5.20 .	194
5.23	Precise architectural model based on the MCS framework, fulfilling figure 5.20	195
5.24	Communication refinement summarized: (a) node-centric, (b) network-centric	196
6.1	The four layer meta-data architecture	201
6.2	Meta-model of ROOM	205
6.3	Notation for unspecified, control and data ports, SPPs, and SUPs .	208
6.4	Notational proposal for a simple CDM	208
6.5	Notation for address spaces	209
6.6	Notation for planes	210
6.7	Conjugation symmetry in ROOM++	210
6.8	Notation for typed binding: (a) binary, (b) n-ary	211
6.9	Notation for a) demultiplexer and b) communication service . . .	212
6.10	Meta-model of (Py)ROOM++	218
6.11	Implementation of four layer meta-data architecture via M2-M0 cascades	223
6.12	Overview of functioning of PyROOM++	224
6.13	Screenshot of a PyROOM++ session	225
6.14	The TCP FSM figure is derived from [Tan96, p.532]. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. User commands are given in bold font.	227
6.15	The FSMs of (a) the vertical and (b) the horizontal interface behavior of the TCP layer. The shortcuts stand for connect and disconnect; the postfixes stand for request, confirmation, indication, and response	228
6.16	The FSMs of (a) the vertical and (b) the horizontal interface behavior of the TCP layer coupled via TDPs and TIPs	230
6.17	The design of the coupled FSM prototype	232

6.18	Screenshot of the coupled FSM prototype	233
7.1	Conceptual schema for system architecture	239
7.2	Grouping of communication entities and the communication service: a) network-centric, b) node-centric	243
7.3	Abstract model patterns for a (a) connection-oriented (CON), (b) connectionless (CNL) communication service	244
7.4	Pattern for communication refinement: a) network-centric, b) node-centric	245
7.5	Conceptual schema for protocol entity	246
7.6	Patterns for a model of the protocol entity; a)-d) represent variations of service and communication interface	247
7.7	Conceptual schema for resource entity	248
7.8	Simple patterns for a) resource user and b) resource provider	249
7.9	Conceptual schema for an aspect	250
7.10	Two aspect entity patterns: a) demultiplexer b) connection management	251
7.11	Overview of SIGTRAN modeling process	253
7.12	Model of the MTP3 user communication network	255
7.13	Model of the MTP3 provider communication network	256
7.14	Model of the MTP3 communication service	258
7.15	Model of the M3UA communication service	259
7.16	Model of the mediator	260
7.17	Views on the SIGTRAN model	261
7.18	Modeling the MGC/MG relation	262

List of Tables

2.1	Mapping of OSI terminology to TCP/IP terminology	37
2.2	Service primitives of TCP as specified in [Pos81b]; “m” indicates mandatory, “o” optional parameters	38
3.1	Mapping of TCP service primitives to service messages	92
3.2	Service messages of TCP separated according to communication types	93
3.3	Service primitives of UDP	101
3.4	Mapping of UDP service primitives to service messages	102
4.1	Mapping addressing conceptions to ROOM run-time conceptions .	146
4.2	Binding addresses to ports: service primitives and service messages	149
7.1	Standards covering case study	252

List of Abbreviations

ABP	Alternating Bit Protocol
ACK	ACKnowledgment
ACME	(name of an ADL)
ADL	Architecture Description Language
ADML	Architecture Description Markup Language
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitive
ATM	Asynchronous Transfer Mode
AXE	(product name)
BSD	Berkley Standard Distribution
BSSAP	Base Station System Application Part
CASE	Computer Aided Software Engineering
CC	Complex Connector
CCITT	Comité Consultatif International de Télégraphique et Téléphonique
CDM	Controlled Domain Model
CEP	Connection End Point
CEPI	Connection End Point Identifier
CHILL	CCITT High Level Language
CLOS	Common Lisp Object-oriented System
CNL	connectionless
CON	connection-oriented
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DIN	Deutsche Industrie Norm
DM	Domain Model
DMS	(Product Name)
DNS	Domain Name System
DPC	Destination Point Code
DSP	Digital Signal Processing
E-CARES	Ericsson Communication ARchitecture for Embedded Systems

EBNF	Extended Backus Naur Form
EED	Ericsson Eurolab Deutschland GmbH
FDDT	Formal Description and Development Technique
FDT	Formal Description Technique
FOCUS	(name of a method)
FSM	Finite State Machine
FSP	Finite State Processes
FTP	File Transfer Protocol
GCP	Gateway Control Protocol
GSM	Global System for Mobile communication
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
HW	Hardware
IANA	Internet Assigned Number Authority
IBM	International Business Machines
ID	IDentifier
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ISUP	ISDN User Part
ITU	International Telecommunication Union
ITU-T	Telecommunication standardization sector of ITU
M3UA	MTP3 User Adaptation
MAP	Mobile Application Part
MCS	Modular Communication System
MDA	Model Driven Architecture
MG	Media Gateway
MGC	Media Gateway Controller
MGCP	Media Gateway Control Protocol
MGW	Media Gateway
MIB	Management Information Base
MML	Man-Machine Language
MOF	Meta Object Facility
MSC	Message Sequence Chart
MTP	Message Transfer Part
NIF	Nodal Interworking Function
OCL	Object Constraint Language
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Object Management Group

OMT	Object Modeling Technique
OO	Object-Oriented, Object-Orientation
OOD	Object-Oriented Design
OOSE	Object-Oriented Software Engineering
OPC	Originating Point Code
OS	Operating System
OSI	Open Systems Interconnection
OSI-RM	OSI Reference Model
PC	Point Code, Personal Computer
PDM	Process Domain Model
PDU	Protocol Data Unit
PE	Processing Element
PI	Protocol Identifier
PLEX	Programming Language for EXchanges
PROTEL	PRocedure Oriented Type Enforcing Language
PTM	Packet Transfer Mode
RCVD	received
RFC	Request For Comments
RM	Reference Model
ROOM	Real-Time Object-Oriented Modeling
RT	Real-Time
RTP	Real-Time Transport Protocol
RWTH	Rheinisch-Westfälische Technische Hochschule
SAP	Service Access Point
SAPI	Service Access Point Identifier
SC	Simple Connector
SCF	Synchronization and Coordination Function
SCN	Switched Circuit Network
SCTP	Stream Control Transmission Protocol
SDL	Specification and Description Language
SDM	Structural Domain Model
SDU	Service Data Unit
SEI	Software Engineering Institute
SEP	Signaling End Point
SG	Signaling Gateway
SI	Service Indicator
SIG	SIGnaling
SIGTRAN	SIGnaling TRANsport
SIP	Service Interaction Point
SMH	Signaling Message Handling
SMTP	Simple Mail Transfer Protocol

SNM	Signaling Network Mangement
SPC	Signaling Point Code
SPP	Service Provisioning Point
SPPI	Service Provisioning Point Identifier
SS7	Signaling System Number 7
STM	Synchronous Transfer Mode
STP	Signaling Transfer Point
SUP	Service Using Point
TCAP	Transaction Capabilities Application Part
TCP	Transmission Control Protocol
TDP	Trigger Detection Point
THE	(name of an operating system)
TINA	Telecommunications Information Networking Architecture
TIP	Trigger Initiation Point
UDP	User Datagram Protocol
UML	Unified Modeling Language
UML-RT	UML Real-Time
UMTS	Universal Mobile Telecommunication System
UNIX	(name of an operating system)
UP	User Part
VM	Virtual Machine
WWW	World Wide Web
XML	eXtended Markup Language

Acknowledgments

This dissertation was part of a research project called E-CARES that had been generously sponsored by Ericsson Eurolab Deutschland (EED) GmbH. During my employment at Ericsson, management showed a lot of confidence and trust in me – I hope the results justify the investment. I would very much like to thank my managers at Ericsson (in chronological order) for their support and belief in me: Ari Peltonen, Andreas Thülig, Per Ljungberg, and Axel Jeske. Thanks also to Kristian Toivo, president of EED, who supported my research.

E-CARES was a cooperation project between EED and the Department of Computer Science III, RWTH Aachen. I am very thankful to my supervisor Prof. Dr. Manfred Nagl for accepting me as an external dissertation student and for the arrangement, which allowed me to finish my doctoral thesis at his department.

At Ericsson, an expert group accompanied the E-CARES project over all the years: Jörg Bruß, Dietmar Wenniger and Andreas Witzel. I owe “my” experts a lot. Their advice was excellent and helpful, and their perseverance was extraordinary. Special thanks to Jörg, with whom I shared an office for quite some time. We had many vivid discussions on modeling and SIGTRAN in specific; it was a lot of fun.

I am also grateful to my former colleagues: Elmar Pritsch, who helped me getting over the first difficulties; Stephan Kruska for all the discussion we had during our billiard sessions and for his contributions to PyROOM++; Thomas Muth, from whom I learned most about what telecommunications really is about – his approach on telecommunications was influential on me. At Ericsson I had the great luck to work together with Stefan Sandh from dpart.com, who was an excellent mentor and became a friend of mine. In the Ericsson context, I met Bran Selic several times; his thoughts on my work were important for me and motivated me to continue my way.

At Prof. Nagl’s department, I also received a lot of support. Many thanks to Bernhard Westfechtel, Manfred Münch, Peter Klein and André Marburger. Moreover, I had the pleasure to work together with Prof. Dr. Manfred Broy, TU Munich, whose algebra was an enlightening source of insight to me. I would like to thank him for co-supervising this work.

A lot of thanks go to my friend Lars von Wedel. Our discussions were a con-

tinuous source of inspiration. Thanks a lot for that! Now we can start working on our unifying theory ;-)

Finally, I would like to express my deepest feelings for my family. Annette, your are a great wife and supported me wherever you could. Little Adrian, you won't remember when you are grown up, but – lucky circumstances – we spent much time together. It is so nice that you are with us!

Chapter 1

Introduction

This work is one outcome of the E-CARES research project,¹ a cooperation project between Ericsson Eurolab Deutschland GmbH (EED) and the Department of Computer Science III, RWTH Aachen [HMJ00, MH01]. In addition, this work was embedded in the post graduate program “Software for Communication Systems” of the RWTH Aachen. The E-CARES project aimed at developing methods, concepts, and tools to support the processes of understanding and restructuring complex telecommunication systems. The project was determined by the combination of two apparently opposite approaches: a top-down approach from a system’s perspective and a bottom-up approach from a “pure” software perspective. The top-down approach deals with the relation between telecommunication standards and system architectures, whereas the bottom-up approach deals with the relation between implementation code and the software architecture. In this work, we describe the result of our investigations on the top-down approach.

Section 1.1 gives a brief introduction into the problem of modeling system architectures in the telecommunication domain. After that, the thesis statement and the approach of this work are explained in section 1.2 and a short overview of the scientific contribution of this work is given in section 1.3. To be concrete, a case study is presented in section 1.4. Later in this work we will come back to the case study and show how we have improved the scenario. Section 1.5 briefly outlines what will come in subsequent chapters.

¹The acronym E-CARES stands for **E**ricsson **C**ommunication **A**Rchitecture for **E**mbodied **S**ystems.

1.1 Problem Description

This work focuses on a very specific area of improvement: *the introduction of architecture models of telecommunication systems and networks in the early phases of system development*. It is the phase in which system engineers design and produce first sketches of new technology to come and to integrate in the legacy. The engineers have to have a system view and see the whole communication network. Their main input are technical standards, the design of the legacy system, and technical studies triggered on own initiative.

Claim: We claim that architecture modeling of telecommunication systems and networks is not yet an established, mature discipline!

This is a provoking statement but gets to the heart of it. Of course, there must be some routine in the development and construction of telecommunication systems. As a matter of fact, the telecommunication system is the world's largest distributed computing system that has ever been built. Most interfaces are standardized, the system design is highly modular, the *plug'n play* principle is more of a reality than in many other computing domains. It constantly evolves and the requirements put on such systems are outstanding: they have to be extremely failsafe (e.g. the GSM standards demands a maximum downtime of a switching center of 30 minutes in 30 years!) and as a consequence thereof, they have to be redundant and maintainable during runtime. Last but not least, telecom systems are real-time systems; they always compute against a deadline. Ideally, these deadlines should never ever be passed even under heavy load conditions. Extraordinary requirements put on the largest and most complex system of the world – so there should be some craft of engineering that has been developed for handling such large and complex systems. So what are we after?

In an established, mature modeling discipline we find (a) a systematic, structured approach to tackle a technical problem (which is also called a *method*), (b) we have means to describe and uniformly communicate problems, solutions and ideas (a *language*), and (c) we have some computerized support that helps in reasoning, simulation, analysis and the like (a *tool*) [Nag96, p.7].

The claim is not that system engineers cannot solve their problems – the claim is that they do not follow a systematic method, that they do not use any stringent (semi-)formal language tailored to their problem domain to describe architecture models, and that they do not create or simulate their models with the help of tools.

Let us have a look at how the work of system engineers in early design phases looks like: Generally, there is a tendency that a design conception has to be proven by implementation; this approach has a long lasting tradition that is deeply rooted in the hacker culture and is a guiding principle for many designers [Ray99]. However, this is not a workable approach for early design phases in an industrial en-

vironment, in which the system engineers experiment with different conceptions and understandings of a, say, new gateway infrastructure. Sitting down and programming the new infrastructure would be extremely time-consuming and delay many answers on urgent questions. Besides that, the target environment for complete networks is not as easily available as interconnected Personal Computers (PCs) are. So, system engineers have developed their own techniques to cope with complexity and level of detail. It is a very intellectual job with almost no tools. They abstract away many details and facts and try to cut down the matter of discussion to the core and solve it. There may be alternative solutions, each of them having its own pros and cons. The argumentation is then documented in form of text and illustrations. After that, the document is handed over to a group of other system engineers. Their task is to inspect the document and see to it whether the argumentation stands their criticism or if alternative ideas may eliminate proposed solutions.

What the author observed in his daily work in the Systems Department at Ericsson is that this process takes much too long time and, because of that, becomes quite costly. The reason is that (a) many discussions are required to get a common agreement of technical understanding and (b) that the documentation of such an understanding is informal, often imprecise, and risks permanent discussions of “what was meant by that statement” and “what do you mean by this in the figure”.

Experience also shows that system engineers have not much use of so-called modeling languages such as the Unified Modeling Language (UML) [BRJ99]. System engineers have to solve problems of distribution, they use abstract concepts like that of a connection, they have to find proper arrangements of protocol stacks, they have to deal with issues of shared resource and remote resource access, and so on and so forth. Since the UML does not offer corresponding concepts and no one has told the engineers *how* UML’s language concepts could be of use anyhow, there is little need for class diagrams, collaboration diagrams and the like. The same is true for SDL (Specification and Description Language) [EHS97] and ROOM (Real-Time Object-Oriented Modeling) [SGW94], two prominent languages in the telecommunication context. It is unclear, how these languages could be utilized for architecture modeling in early phases.

The author does not believe that his experience and observations are specific to Ericsson. There are no courses to attend nor are there any books available that teach how to improve and make it right. Take for example the well-known title “Computer Networks” from TANENBAUM, now in its 4th edition [Tan03]. TANENBAUM is a talented writer who understands to explain and teach a complex and complicated topic in simple terms and narrative voice. The theme, computer networks, is methodically structured and presented. However, TANENBAUM has no method to model and present networks and system architectures. He uses the same style of architecture “models” we are about to criticise.

The aim of this work is to contribute to the goal to equip systems engineering with instruments (method, language, tool) of an established and mature modeling discipline.

1.2 Thesis Statement and Approach

1.2.1 Thesis Statement

Thesis Statement: In this work we present a systematic approach to create logical models of network architectures of virtually any telecommunication system. The thesis is that such an approach can be based upon as few as three basic cornerstones: the *types of communication* and the design principles of *distribution* and *layering* in a network system.

First, what do we mean by “logical models of network architectures”? Here, we are having a very specific meaning of the word “model” and the activity “modeling”. In natural sciences a model represents a detail of the “real” world; something that is a simplification of something observable. The model represents a hypothesis about measurable facts; it can be thus falsified (see [Pop94] but also [Cha94, p.63 ff.]). This is not necessarily the case for models created in computer science. Here, a model is foremost a description of abstract conceptions and their interrelations. In the second place, a model might represent an observable cutting of reality. Ergo, models in computer science cannot always be falsified, they cannot proven to be wrong or right. Criteria that are used instead to judge about a model are its ability to solve a problem, its plausibility, its usefulness, its meaningfulness, but also implied measures such as predicted implementation costs, extensibility, maintainability etc.

A model can be expressed in various sorts of languages and in various styles. A language can be informal, semi-formal,² or formal³; the notation can be textual, graphical or both; the language can be adapted to the problem domain or not. For this work, we strive for models expressed in a formal language with precise syntax and semantics. The presentation style, textual or visual, is a secondary but not unimportant issue. Throughout this work we prefer a graphical notation for the obvious reason that visual models are easier to grasp. But one should not expect too much of visual models; certain details can be much better expressed by structured text.

By “logical models” we mean models that consist of logical entities that abstract away the fact that this entity may stand for a piece of hardware or software; it is rather the assigned functionality that qualifies the entity. If an entity is labeled as a “coax cable”, the entity depicts a physical device – it may be realized by a

²The UML is a semi-formal technique since it contains constructs with unclear semantics. In specific, the UML lacks precise execution semantics.

³One can further distinguish Formal Description Techniques (FDTs) from Formal Description and Development Techniques (FDDTs). FDDTs goes beyond FDTs in their support for logical deduction of implementations from specifications ([BS01, p.9], [AP98, p.39]). SDL and ROOM belong to the class of FDTs.

real coax cable or emulated by software, we do not care. What we are after is to work with abstractions. It is the conception with its properties that counts. In that respect the models belong to the domain of systems engineering rather than to the domain of software engineering.

To continue: What is meant by “network architectures” in the thesis statement? The *architecture* of a technical system reflects significant design decisions about the system’s organization and is typically documented as a description of key system elements (be they hardware or software), their composition, functioning and interrelation. Possibly, several viewpoints on the architecture are required to make the description complete. This description is usually sort of high-level and abstracts away a lot of details. The process of creating a description of an architecture is also called *architecture modeling*.

In academia, there is no commonly agreed definition on architecture, so our definition just given can be questioned, but we find ourselves in agreement with the main stream. The Software Engineering Institute (SEI) lists numerous definitions on its web site⁴ with most definitions taking a similar, largely structural perspective on architecture.

Pragmatically, we follow the standpoint of CLEMENTS et al. that the architect defines what the architecture is [CBB⁺03]. This standpoint is very much reality in a domain, in which standards rule development. Very often it is the standard that states what is to be regarded as the system’s architecture and what is not. Insofar, we are not in need to make up our minds what the “significant design decisions about the system’s organization” are. Architecture is design, but design is not architecture – where to draw the borderline is subjective and usually given by definition or authority. What is more of a concern is which viewpoint we take on architecture in order to model it.

In this work we see architecture from a C&C, component and connector, type of view. We consciously neglect other viewpoints such as the allocation viewpoint in order to limit the scope of this work. Other important parts of a complete architecture model like e.g. a design rationale [Nag90] are neglected for the very same reason. However, it remains to clarify what we mean by the component and connector concepts. With the decision to take ROOM as a modeling language the distinction between a module and a component is blurred, which makes a precise delimitation difficult. By definition, we declare ROOM’s actor classes as components and ROOM’s bindings as connectors. More on ROOM and these key concepts of our architecture viewpoint can be found in chapter 2.

The addition “*network architecture*” in the thesis statement highlights that the architecture level we consider includes components of a network, which are distributed and often physically separated in space. So, remote communication be-

⁴See <http://www.sei.cmu.edu/architecture/definitions.html> (2003-06-14).

comes an important architectural issue. Note that we use the terms “system architecture” and “network architecture” interchangeably.

So far the explanatory comments on the thesis statement.

1.2.2 Approach

The thesis statement is quite demanding: If we claim to have a systematic approach to virtually model *any* telecommunication system, we owe the reader a proof of statement. Directly said: that is impossible. Rather we would like to persuade the reader by a double-track strategy: (1) Generic track: We analyze the basic design principles underlying each and every distributed communication system and derive a set of generic solution patterns from that. As is mentioned in the thesis statement, we believe it to be sufficient to analyze the consequences of *distribution*, the effects of *layering*, and the *type of communication* in communication networks. (2) Concrete track: To substantiate our argumentation the solution patterns are exemplified on concrete examples. The examples are chosen in such a way that they lay the basis for a larger case study, a complete architecture. Thereby, we show step by step how a real network model can be composed of some few patterns. The case study is of modest complexity (but quite a challenge to model) and contains a lot of problems we have to generically solve for every network. This way we demonstrate the applicability of our approach on real-life engineering problems.

For (1), the generic track, we use an algebra to precisely define basic principles in communication networks, and ROOM as a modeling language to describe generic solution patterns. The decision to take ROOM is an *a priori* decision. It is based on the insight that ROOM (a) has its origins in telecommunications and (b) qualifies as an Architecture Description Language (ADL). More on ROOM and our reasoning of choice can be found in chapter 8.

For (2), the concrete track, we introduce a case study in section 1.4. The purpose of this case study is twofold. First, the case study is a short introduction to SIGTRAN (SIGnaling TRANsport) with material directly taken from the standards. Insofar, the case study gives an impression of how standards “model” architectures. We will thoroughly discuss the problems and curiosities we note in dealing with standards. Second, the case study serves as our main example of reference. With the techniques developed in this work we will demonstrate how one can do a better job. We will present a new model of SIGTRAN that can be compared with the standard. It is therefore a good example of how to move *from standards to system architectures*.

When we speak of a “systematic approach” in the thesis statement, we indicate that one focus in this work is on the *method* part. That is what most system engineers suffer from, the lack of a proper modeling method. However, the dis-

cussions in the method part also unveil insufficiencies and deficiencies of our *a priori* language ROOM. As a consequence thereof, another important part is to equip our method with a proper *language*. We do this by proposing extensions to ROOM. Finally, the *tool* part is a straight implementation of the new, extended ROOM language in a tool prototype. Given all these three parts – method, language, tool – we have all building blocks together to lift architecture modeling of telecommunication systems and networks to a mature discipline.

1.2.3 A Note to the Reader

For this work it is advantageous if the reader is well acquainted with data communication, distributed networks and telecommunication. We assume a background that comprises the knowledge of computer networks as presented e.g. in [Tan03, Hal96], of distributed networks as introduced e.g. in [TvS02, CDK01] and of telecommunications as presented in e.g. [Rus00, Stu97, Stu98]. Furthermore, the reader should be familiar with “architectural thinking”; basic textbooks in that respect are [BCK98, SG96, Nag90, HNS00]. Ideal is also some background in the following languages: SDL [EHS97] and/or ROOM [SGW94].

1.3 Scientific Contribution

Technically, the field of communication systems is well understood – but the discipline of modeling such systems is not. We will not contribute anything to the technical domain: we do not invent new technology, we do not improve routers or switches, we do not optimize communication protocols. What we contribute to is the understanding of the overall: not to get lost in the details of all the bits and pieces that make up a communication system; to see a network as a complete system that can be systematically decomposed into its constituting elements; to work with suitable abstractions of the system on precisely defined views; to support architecture thinking and reasoning on a network level. The systematics presented enable even a newcomer in a network domain to catch up with the experts in the field in quite short time. For example, the author and a former colleague spent roughly one year almost full-time on SIGTRAN until they had confidently understood all the technical material. Lots of discussions with other colleagues were needed including people from standardization to clarify many issues. If the material had been presented in a way this dissertation promotes, the process could have been reduced considerably – which is a relevant aspect in a business context. It was also experienced that this new clarity in system design releases creativity. Thus, as a side-effect, new and improved technology can be the outcome.

From a scientific viewpoint, unique for this work is

- the development of a method to create architecture models out of generic patterns;
- the postulate that such a method can be based upon as few as three basic cornerstones: the design principles of *distribution* and *layering* and the *types of communication*;
- the formalized argumentation and the mathematical reasoning especially about distribution and layering;
- the systematic approach in structuring the domain of distributed, layered communication systems, which is novel;
- the identification of missing/suitable constructs in today's modeling languages to depict architecture models for communication systems (exemplified on ROOM);
- the definition of new concepts to extend ROOM;
- the implementation of a new sort of modeling tool that enables rapid model prototyping.

In short, this work aims to inaugurate architecture modeling of telecommunication systems and networks as an engineering discipline.

1.4 Case Study: SIGTRAN

1.4.1 Background

It is a challenging task to find a case study that is not too complicated but also not too simple to use as an example for architecture modeling. Just by coincidence, the author and some of his former colleagues were assigned to study and investigate signaling transport on IP-based network technology. It proved to be an excellent assignment by management, since SIGTRAN contains almost all sorts of problems one could be faced with in network design, which made us revise the method and the language presented in this work over and over again. What makes SIGTRAN so interesting is that it lies on the border of two different technology domains: the traditional signaling network technology of telecommunications and the IP-based network technology of the Internet. So, we could also verify if our conceptions embrace both domains. This is an important aspect since both technologies grow more and more together.

The SIGTRAN standards are developed under the head of the Internet Engineering Task Force (IETF). The standards are published as so-called RFCs (Request For Comments) and are publicly available on the Internet.⁵

1.4.2 The SIGTRAN Architecture

Leading to SIGTRAN

Simply speaking, the architecture of a modern telecommunication network consists of at least three logically separate but collaborating networks, also called planes: a user plane, a control plane and a management plane. The *user plane* transports the payload, the actual data, be it speech, music, images, animations, raw binary digits, whatsoever. The *control plane* transports signaling information; that is why the control plane is also often called the signaling system. Signaling information is needed to (a) control resources of the user plane (such as converters and switches) and (b) to coordinate and supervise the use of user data resources in order to set up a voice connection or a video stream, for example. The *management plane* is used to configure and manage both, the user plane and the control plane.

Let us illustrate the interaction of the user and the control plane on a simple use case: a mobile call. When a user dials a number on a mobile phone and presses the “lift off” button, first the control plane comes into action. Based on the dialed number the signaling system queries databases to locate and identify

⁵see <http://www.ietf.org>

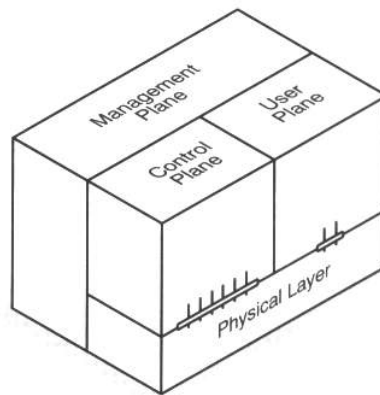


Figure 1.1: The ISDN Reference Model; taken from [EV98, p.117]

the resources which are in charge of the addressed device; usually, another mobile phone or a fixed-line termination. The resources typically involved in such a scenario are radio base stations, switching centers, gateways and service centers. Then, the signaling system tries to pre-allocate devices (which we also neutrally call “resources”) within the base station, the switching centers etc. to set-up the user plane. The resources could be a radio channel, time slots (in case of STM, Synchronous Transfer Mode), packets (in case of PTM, Packet Transfer Mode) or cells (in case of ATM, Asynchronous Transfer Mode) [Sie97, p.17ff.], speech en-/decoders, terminations in a switch, echo cancellers and so on. The control plane reserves and configures a whole chain of resources for the transport of user data. If all resources are successfully pre-allocated and if the callee accepts the call, then the payload can flow over the user plane via the allocated resources. During the call, the control plane continues to supervise the call. If one of the parties of the call hooks on or if the radio contact is lost, the signaling system tears down the call and releases all involved resources.

From this use case we can understand, why system engineers put so much focus on the control plane, which is the logic of the network and in charge of almost everything, and often neglect the user plane. The management plane is often not considered in the first place and has its own problem domain; we will mostly drop management plane related issues.

The distinction in three planes was introduced with the advent of ISDN (Integrated Services Digital Network) [ITU93a] and shaped the architecture of many subsequent network architectures like GSM (Global System for Mobile communication) [EV98] and UMTS (Universal Mobile Telecommunications System) [WAS01]. The basic schema is sketched in figure 1.1. Physical resources belong to the physical layer.

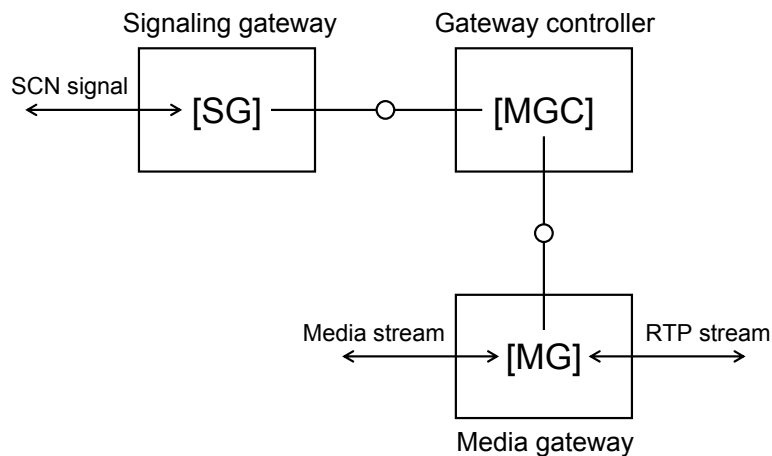


Figure 1.2: SIGTRAN Functional Model; a simplified version of figure 1 in [ORG⁺99]

A more detailed introduction into basic architectural principles of telecommunication systems is given in chapter 2. Here, the rough distinction of a telecommunication architecture in planes should help the reader get into our main example of reference throughout this work, the SIGTRAN (SIGnaling TRANsport) architecture.

The SIGTRAN Functional Architecture

The convergence of traditional telecommunication and Internet technology has led to new solutions in the integration of both technology domains. One of these solutions is the framework architecture for signaling transport, SIGTRAN for short, in the UMTS system.

The SIGTRAN architecture [ORG⁺99] builds a link to interconnect a (traditional) telecommunication network and an IP-based network. Mediation between different networks and communication infrastructures is generally achieved by so-called gateways. Generally speaking, a gateway converts and adapts the communication protocols from one technology domain to another technology domain; this includes also the conversion of address spaces and schemes and the translation of management messages. Consequently, the functional “model” of the SIGTRAN architecture is built up of gateways, see figure 1.2. Note that all elements in figure 1.2 are functional entities; they may be but need not be implemented as physical entities.

There is one gateway for the conversion/adaption of control plane information, the Signaling Gateway (SG), and another gateway for the conversion/adaption of user plane information, the Media Gateway (MGW). The signaling gateway does

not directly control the media gateway; instead, there is a special function, the Media Gateway Controller (MGC), that controls the media gateway.

The signaling gateway processes SCN (Switched Circuit Network) signals from the telecom side and some sort of IP-based signaling information from the IP network side. The media gateway processes a media stream from the telecom side and an RTP (Real-Time Transport Protocol) [SCFJ96] stream from the IP network side. The signaling traffic between the signaling gateway and the gateway controller, and between the gateway controller and the media gateway is also IP-based.

Be aware that we intentionally only use figures from the SIGTRAN standard in this section; we just polished the drawings a little bit for this work. Besides introducing SIGTRAN technology, the purpose of this section is to give an idea how standards and related documents “model” the domain. That is the reason why we do not modify the figures even if it might be helpful for understanding purposes. We will discuss our observations later in section 1.4.3.

The SIGTRAN Protocol Architecture

The SIGTRAN architecture aims to provide means for the “transport of message-based signaling protocols over IP networks” [ORG⁺99]. So the official statement, but what is meant by this quote?

For the control plane a signaling system called Signaling System Number 7, also abbreviated as SS7, has become *the* standard in telecommunication networks [ITU93b]. SS7 consists of an infrastructure of Signaling End Points (SEP) and Signaling Transfer Points (STP) and a set of signaling services. The services can be divided into a Message Transfer Part (MTP) and several User Parts (UP). MTP realizes basic means for message transfer in SS7, UPs are used directly by user applications and provide higher-level services. Common user parts are e.g. ISUP (ISDN User Part) for ISDN services and TCAP (Transaction Capabilities Application Part) for database transactions. For mobile networks there are some specific extensions like for example MAP (Mobile Application Part) and BSSAP (Base Station System Application Part).

There are numerous telecom applications that rely on the existence of an SS7 signaling system or – more specifically – on SS7 user parts. Huge investments have been made to develop and deploy these applications. The key idea of SIGTRAN is reuse: reuse as much as possible. To let the user parts also run on an IP-based network, SIGTRAN provides an MTP-like interface on the IP side so that all user parts can be just stacked atop that interface without any modifications. In other words, the user parts run completely independent of the actual message transport mechanism being used. The user parts do not notice if MTP or SIG is used as a reliable signaling transport function, see figure 1.3 a) and b).

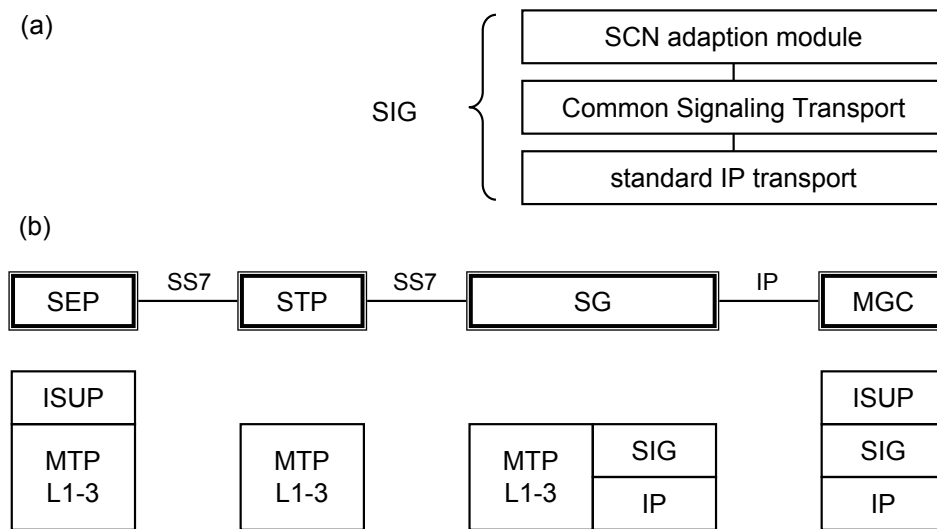


Figure 1.3: SIGTRAN Signaling Transport Components and Protocol Architecture; see figure 6 and 7 in [ORG⁺99]

SIG is generically decomposed in a standard IP transport component, a common signalling transport component, and an SCN adaption module component, see figure 1.3 a). SIG sits on top of IP and provides an interface identical to the interface provided by a traditional MTP Layer 3, see figure 1.3 b). Although functionally equivalent from a user part point of view, SIG is in contrast to the MTP protocol stack. The three layers of MTP have the following functions: “Level 1 defines the physical, electrical and functional characteristics of a signalling data link and the means to access it. [...] Level 2 defines the functions and procedures for, and relating to, the transfer of signalling messages over one individual signalling data link. [...] Level 3 in principle defines those transport functions and procedures that are common to, and independent of, the operation of individual signalling links. [...] [T]hese functions fall into two major categories: (a) Signalling message handling functions⁶ and (b) Signalling network management functions⁷” [ITU93c]. It is important to note that the SIGTRAN framework architecture does not further break down the SIG components but rather formulates requirements on SIG to fulfill. The functioning of the SIG components may thus look completely different from the MTP layer functions.

⁶“These are functions that, at the actual transfer of a message, direct the message to the proper signalling link or User Part.” [ITU93c]

⁷“These are functions that, on the basis of predetermined data and information about the status of the signalling network, control the current message routing and configuration of signalling network facilities. In the event of changes in the status, they also control reconfigurations and other actions to preserve or restore the normal message transfer capability.” [ITU93c]

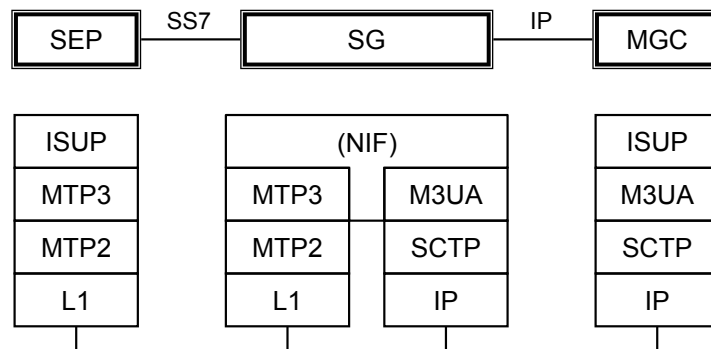


Figure 1.4: SIGTRAN with M3UA: ISUP Message Transport; taken from [SMPB02] (example 1) with slight modifications

Since the STP and the SG act only as a transfer point and as a converter, respectively, the end points host the user parts. In figure 1.3 b), ISUP exemplifies the user part for the SEP and the MGC. The two user parts can interact with each other without any notice that one resides on an traditional SS7-based network whereas the other resides on an IP-based network.

SIG Exemplified on M3UA

A concrete specification of SIG is given by the MTP3 User Adaptation Layer, M3UA for short, see [SMPB02]. M3UA relies on the Stream Control Transmission Protocol (SCTP) [SXM⁺00] as a layer over IP. SCTP is very close in functionality to the well-known Transmission Control Protocol (TCP) [Pos81b] but with certain improvements and is promoted as a potential replacement for TCP in the future [SX01].

With M3UA and SCTP as incarnations of the SIG components, the “new” protocol architecture looks like as shown in figure 1.4. New is the Nodal Interworking Function (NIF), which is not at all mentioned in the SIGTRAN framework and is essentially left unspecified in the M3UA standard. We can just deduce that NIF stands for some functionality that is neither offered by M3UA nor MTP3 but required for the interwork of both.

1.4.3 Problems and Curiosities

What we have seen in the previous subsection is not only an explanation of SIGTRAN but also a typical example for the way how information is presented by technical standardization bodies: plain english text with some supporting drawings. That’s it! Totally document driven, no “real” models, just “pen and paper”.

In the SIGTRAN example we referenced some few technical standards from the Internet Engineering Task Force, IETF, and the International Telecommunication Union, ITU. The reader may inspect other technical recommendations from IETF, ITU or any other standardization body; the outcome will be the same: for each technical recommendation (or set of recommendations) an own conceptual domain space (more formally called an *ontology*) and style of presenting/modeling the topic under discussion is invented. While this is perfectly all right, the problem is that neither a methodological approach in analyzing or constructing network and system architectures is followed nor is there any consequent notational formalism applied for modeling such systems. In the cases in which formal languages are used such as SDL, the Specification and Description Language [EHS97] (SDL is a very prominent language in telecommunications), their usefulness is often restricted to behavioral specifications.

But what exactly do we complain about in the SIGTRAN example? Let us reinspect the SIGTRAN example and especially focus on the figures 1.2–1.4. What is odd? What is inconsistent? What remains unclear?

Notes about the SIGTRAN Functional Model

Functional models present logical entities and their relationships among each other. Most likely, functional models are one of the oldest engineering tools. They range from sketches on the functioning of a technical device up to very formal, abstract information models, such as entity relationship diagrams.

The use of functional models is problem-free as long as some syntactical and semantical conventions are followed and as long as these conventions lead to a consistent presentation, which is free of discrepancies and unclear meanings (at least to a certain degree). However, the figures in standards and in many other technical documentation very often remind of drawings on a drawing board. No conventions have been agreed upon, symbols rely on common sense, the interpretation is vague, often intuitive. Usually, these sorts of drawings just visually support a textual explanation. Nonetheless, system designers and architects do not hesitate to call such figures “architectures” or “models” (as is indicated by the captions) and are often very proud of the pretended preciseness of the figures and the level of technical detail. They can be easily proven to be wrong as demonstrated on figure 1.2.

This said it is not surprising that even a simple figure like the SIGTRAN functional model (figure 1.2) can raise a bunch of questions. Neither the author with his background in SIGTRAN nor other experts in the domain could clearly answer or argue about the following questions and observations:

- It is unclear why we seem to have two levels of components: The arrows do

not terminate at the outside of a functional component box but at something inside.

- Some lines have arrow heads, some do not have arrow heads. Is there a reason for that?
- What is the meaning of the lines connecting the functional components at all? Do they represent connections or just logical relations of data flow?
- The meaning of the “o” in the middle of some lines can only be guessed: It seems to indicate “interfaces” which are addressed by SIGTRAN; but is that interpretation correct?
- The media gateways, the gateway controller and the general set-up are not unique inventions of SIGTRAN. It remains unclear why this is called a “functional model” and what is SIGTRAN specific in this model.
- The distinction in a user and a control plane is graphically not supported and cannot be deduced from the figure.
- The gateways are located at the borderline of two different network technologies. However, in the figure we see two arrows, the SCN signal and the media stream, pointing into emptiness. There is no notion of connected networks. Why not?

Notes about the SIGTRAN Protocol Architecture

So-called protocol architectures are a very common and widespread way to present a whole network system or parts thereof with all key components at a glance. They are favorite since they allow an engineer to give a complete overview of the protocols and the nodes hosting these protocols on a single sheet of paper. Even large systems like GSM or UMTS easily fit on a DIN A4 or DIN A3 printout.

Drastically speaking, some system architects stop working when they successfully completed a protocol architecture. Together with some supplementing notes about the functions to be fulfilled by the nodes in the network, everything else is regarded as detail work. While this is surely an oversimplification, it emphasizes the importance and the relevance of protocol architectures in the daily work of system engineers.

Anyhow, protocol architectures also lack consistency and clarity and are open to misunderstandings and misinterpretations. The SIGTRAN protocol architecture (figure 1.3) is a very good case study for the nuisances in protocol architectures in general:

- The figure seems to relate logical functions and the protocol stacks related to these functions. Still there is the question whether e.g. the MGC is merely a logical container for the IP/SIG/ISUP protocol stack or if the MGC functionality goes beyond that?
- The functional entities are connected by lines with SS7 and IP as labels attached to them. What is the precise meaning of these lines? If they represent technologies the connection is based upon, the meaning of an IP connection is somewhat awkward, since IP is connectionless by nature.
- What happens in the signaling gateway? We just see two protocol stacks side by side, but how is the interworking done? Why is there no entity that comprises the interworking function?
- MTP level 1 and level 2 address the physical and data link layer. However, the physical and data link layer underneath IP are not shown. Why not? It makes the figure inconsistent in the level of detail presented.
- The figure strongly implies that there is a relationship between the levels of MTP and IP/SIG. Nothing could be more misleading. The MTP stack and the IP/SIG stack can be hardly compared in functionality. Many authors have tried to do so in similar cases, but the result is always debatable. The key point is that the SG integrates technology domains and does not try to map protocol functionality.
- As was mentioned, for the user parts the underlying network technology becomes transparent. Does that key point of SIGTRAN become apparent from looking at the protocol architecture? To a certain degree, yes, since the STP and the SG lack a user part; but it requires background knowledge about SS7 and cannot be solely concluded from the figure.
- Since SIGTRAN aims at transparency on the message transfer part, everything above the MTP3 interface is pure SS7 technology. In that sense, the MGC can be regarded as another SEP. That role of the MGC is neither clear from the figure nor is it clear from the standard [ORG⁺99].
- For some reason, an STP is shown in the figure. There is a point to make that routers in an IP network may be comparable to the status of STPs in SS7. Again, this is an inconsistency on the presentation level. Is the existence of STPs and routers of any value for the explanation of the SIGTRAN protocol architecture?
- The relation MGC/MGW is not further addressed by SIGTRAN. So, what do we do about it?

Notes about SIGTRAN with M3UA

Figure 1.4 is another evidence for the immature discipline of modeling telecommunication systems. What is essentially new in the SIGTRAN protocol architecture as provided by the M3UA standard, is the Nodal Interworking Function (NIF). NIF is a characteristic function of relays and gateways but often there is no clear indication given how a protocol and NIF interrelate. Some issues we can address are:

- NIF is a new component in the SG and non-existent in the SIGTRAN framework. An issue that requires clarification, which is neither given by SIGTRAN nor the M3UA standard.
- Why is NIF embraced by round brackets? What is the meaning of this notation? Is it because NIF is not a protocol but an internal function (what it in fact is)?
- Since NIF is visually placed on the same level as ISUP is, the “innocent” reader may interpret NIF as a protocol complementing ISUP. The figure supports such a misinterpretation.
- For what reason is MTP1 now noted as L1 (Layer 1)?
- In the figure there is a line that connects the L1 layer and another line that connects the IP layer. What is the meaning of these lines? For the L1 layer the line may stand for a physical connection, but for the IP layer we are not on the level of physical connections.
- In which respect do the interconnections of the protocol stacks relate to the interconnections of the functional components at the top of the figure?
- It looks very much like that M3UA and MTP3 are on the same functional protocol level, and so do SCTP and MTP2, IP and L1. This comparison is full of pitfalls and basically wrong.

Forgotten Issues

There are some issues in system architecture modeling which are important but rarely made explicit. That does not mean that these issues are not properly understood. Rather, the author assumes that systems engineers do not know how they should handle these issues in their models.

Addressing Addressing is one of these issues. For example, in SIGTRAN a conversion between Signaling Point Codes (SPC) and IP addresses takes place at the signaling gateway. However, on higher levels we only work with SPCs or Global Titles etc. depending on the user part. How is that modeled?

Resources Another issue is resource handling. Protocols alone do not do so much. Somehow and somewhere physical or logical resources must be controlled and used. Without a firm understanding of user/resource relationships, communication architectures cannot be modeled properly.

Quality of Service Quality of Service (QoS) is a further issue. QoS attributes can be rarely found in system models even though their importance is unquestionable; they may substantially influence the system architecture design. For example, demands on system reliability may lead to redundancies in form of duplicated entities. The question is where to assign QoS attributes to in system models.

In the course of this work, we will come back to these “forgotten” issues and pay attention to them.

1.5 Overview

In the following we give a brief overview of the chapters to come.

Chapter 2: Foundations

The foundations for this work are being laid in the next chapter, chapter 2. We will start with the most commonly used and referred to framework in telecommunications, the Open Systems Interconnection Reference Model (OSI RM). In addition, we will also consider another widely spread framework, the TCP/IP (Transmission Control Protocol/Internet Protocol) RM; TCP/IP is the other popular RM primarily used in the Internet technology domain. We will diagnose that OSI as well as TCP/IP favor a dominant view on communication systems, which limit their use and applicability for modeling such systems. So, the goal of the first step is to establish an unbiased view and work out core principles of both reference models. As a result of these efforts, we will present the structuring of a communication system in a horizontal and vertical dimension as equally important and with equal rights: neither *distribution* (horizontal structuring) nor *layering* (vertical structuring) is per se superior or inferior to the other.

A further basic insight originates from dealing with real-time systems: a real-time system basically is a reactive system, and a reactive system requires some sort of internal representation, a model of its environment in order to interact with the “outer” world – at least, when the reactive system aims to exert control, meaning when it aims to influence the outer world in a meaningful manner. This understanding of modeling the outside world leads to the term of a Controlled Domain Model (CDM). It is the aspect of control in a communication relationship, which lets us distinguish different *types of communication*.

Herewith, we are having the three basic cornerstones required for an elementary understanding of complex communication networks: *distribution*, *layering* and *types of communication*. These cornerstones are the main themes of the next three chapters.

Before that, we give a brief introduction to the modeling language of our choice, namely ROOM (Real-Time Object-Oriented Modeling). In ROOM, a system is specified in a component and connector (C&C) style. The components, called actor classes, can be composed of other components, realized through so-called actor references, which point to the respective actor classes. Components can be interconnected via bindings. The possibility to label an actor reference as fixed, optional, substitutable or as a placeholder for an actor class instance being imported at run-time, makes the language very powerful and expressive for modeling component-oriented architectures.

To also provide a formal foundation for this work, a component-oriented algebra, called FOCUS, is being introduced, which enables a precise description of the core principles of distribution and layering.

Chapter 3: Types of Communication

In chapter 3 we will investigate elementary types of communication relations. The starting question is: What is control? We will define control as the directed exertion of influence on the behavior of the communication partner. As an immediate conclusion, we can state that the behavior can be meaningfully directed only if one knows (a) the current state of the controllee (at least to a certain degree of probability) and (b) with which measures, called stimuli, one can provoke which sorts of directly or indirectly observable activities (also with a certain degree of probability). Ergo, it is required to have a state model – or more generally – to have an internal representation, simply called model, of the other party. This model is the so-called Controlled Domain Model (CDM). The CDM just describes that fraction of the communication reality as it is observed by the controller. The CDM is not a complete model of the controllee, but the controller's viewpoint on the controllee.

The question “Who controls whom?” can usually not be answered by an external observer, who monitors the exchange of messages between two communicating parties. One needs knowledge about whether one party has a CDM of the other party to exert control or not. In our architecture models, we regard the aspect of control as vital and make the CDM explicitly available. We therefore argue for grey-box specifications of controlling components. In our models, the CDM will be made public as a model artifact placed on a component's border. Further internals of a component remain hidden from an external viewpoint.

We can distinguish three basic combinations of the exertion of control between two communicating parties: (a) no side exerts control, i.e. no side has a CDM to influence the other side's behavior in a controlled way, which we call *data-oriented* communication; (b) only one side exerts control, which we call *control-oriented* communication; (c) both sides exert control, which we call *protocol-oriented* communication. The first two cases, data-oriented and control-oriented communication, could be related to the traditional distinction in data flow and control flow. The last case, protocol-oriented communication, is a remarkable exception and seems to be a unique specialty in communication systems: both parties try to control each other in order to achieve a common goal. Mutual controlled communication must be a special sort of communication that has to struggle with the communication as such, i.e. with communication that is subject of disturbances, interferences and so on. Mutual control is the attempt to compensate these disruptive elements by a set of rules, binding for both parties, how to normally react on the receipt of messages and how to react on exceptions on the expected flow

of information. In telecommunications, such a set of rules is called a *protocol*. This notion of a protocol can be distinguished from the also common convention, to define the term “protocol” as a set of possible messages to be exchanged. We prefer to call the latter meaning a *message schema* and use the term *protocol* for a message schema that is enriched by a set of rules for communication.

To sum up, the aspect of control, the motivation for grey-box specifications, and the distinction of types of communication are at the heart of chapter 2 and are exemplified on three case studies: on modeling TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and MGCP (Media Gateway Control Protocol).

Chapter 4: Distribution

The notion of distribution refers to the physical and spatial separation between two communicating parties in space. For communication, the distance in space has to be bridged by a communication medium. This medium is, depending on its properties, in most cases non ideal and has a disturbing impact on the communication. The exchange of messages via the communication media may result in a loss of messages, may cause defects in messages, may impact the sequence of messages (e.g. some messages may overtake other messages) etc. All these effects can be condensed in a model of the communication medium, the so-called *complex connector*. Thereby, the physical aspect of distribution is logically modeled by an abstraction of a communication media. The notion of a complex connector is the key issue of chapter 4.

Since remote communication can be either connection-oriented or connectionless, these are the basic cases to consider. Connection-oriented communication can be further characterized as static or dynamic. Connection-oriented communication is static, if there is by default a connection established between two communication parties, which cannot be removed. This sort of communication is – per definitionem – undestructable and only in rare cases an adequate abstraction for a “real” communication channel. Normally, connection-oriented communication is dynamic: a connection to another party is requested, established, and – later on – released. For that purpose we need a communication service, who maintains and controls connections. To be methodically strict, the communication service can be interpreted itself as a complex connector, usually an n -ary complex connector. For pragmatic reasons, we leave it up to the modeller to alternatively define a component in form of an actor class as the communication service.

In connectionless communication, messages are delivered to their destination without setting up a connection; ergo, there is no connection management needed. Here, the connectionless communication can be also methodically correct modeled as a complex connector offering a distribution service instead of a connection

service. Optionally, one can also represent the distribution service via a casual component.

In chapter 4, connection-oriented and connectionless communication are thoroughly discussed on the example of TCP and UDP; TCP is connection-oriented, UDP connectionless. The importance of the notion of a complex connector is reflected by the extension of ROOM's binding concept to the concept of a *typed binding*.

The difference between connection-oriented and connectionless can be also understood as a fundamental difference in the way, messages are being addressed to their destination. A connection can be seen as an special form of an address: the connection as an alternative representation of a previously determined destination. In the connectionless case, the destination address has to be specified for each and every message; choosing a substitute representation is no option. This sort of considerations emphasize the meaning of addressing and the need to model and structure address spaces. The relevance of the topic also leads to suitable language constructs in ROOM.

Chapter 5: Layering

So far, we are capable to model distributed systems, but one of the most important design principles in communication systems, the principle of *layering*, is not covered, yet. Chapter 5 is all about this topic. Having the conception of a complex connector, layering can be resolved as the replacement of a complex connector by a distributed system. The complex connectors of a distributed system are themselves distributed systems. Consequently, the complex connector is an abstraction of an "underlying" distributed system. This form of layering can be sharply distinguished from the structuring principle, in which a "lower" layer provides services to an "upper" layer; the "upper" layer may in turn provide services to the next layer above. While layering with complex connectors truly leads to an abstraction hierarchy – we call it also communication refinement –, layering as an organization of service providers and users is only characterized by a separation of life-time contexts and sort of "uses" relations.

Layering in the sense of communication refinement allows an system architect to have two viewpoints on a system model: a node-centric and a network-centric viewpoint. The node-centric viewpoint corresponds to the traditional understanding of layering in a communication network and is manifested by the focus on protocol stacks in a node. Within such a node, the scope is narrowed to layers as service providers and users, here protocol-oriented communication services. This viewpoint is typical for an implementation-oriented perspective, which sees nothing but the node. This perspective is limited and partially misleading, since it hides the view on the system as a whole. A network-oriented viewpoint, on the

other hand, perceives a communication network as an infrastructure of “nested” distributed communication systems. This is a typical viewpoint of a system architect. However, this viewpoint has not made it as a consequent design principle in the canon of (intellectual) tools available for modeling complex communication systems. Most likely, the reason is that OSI RM and TCP/IP RM are very much node-centric on their viewpoint on the world of communication systems. OSI and TCP/IP are so much dominant that the attempt to change viewpoint is almost identical to a paradigm change. But as a matter of fact, both viewpoints (node-centric and network-centric) are of equal right and can be transformed to one another.

The different meanings of layering and the different viewpoints of communication refinement are discussed in detail on the example of ROOM and are made concrete on TCP, UDP and MGCP. In the course of our investigation, we will realize that horizontal and vertical structuring of communication systems are relative terms and require a reference point. Furthermore, we will see that the concept of a complex connector can be used for more than abstracting a “lower” distributed system. A further insight concerns the use of planes in communication systems. Planes are spheres of functional use and can be respected by the introduction of namespaces in ROOM.

With all the described techniques we are having a powerful set of conceptions at hand, to model distributed, layered communication systems.

Chapter 6: Language and Implementation

All the extensions we made to ROOM throughout the previous chapter are summarized in chapter 6. The extended ROOM language is called ROOM++, its implementation in the programming language Python, PyROOM++. ROOM++ is stringently designed and implemented as a meta-model in a four layer meta-data architecture. The focus is on the realization of architecturally relevant structural conceptions; behavior has to be specified as a Python program, the use of state machines is not supported but can be easily added via a library. For this case, we propose an extension to state machine modeling, which enables to decouple functional aspects of a “large” state machine into smaller state machines and synchronize their coordination and cooperation via so called trigger detection points and trigger initiation points.

The implementation of ROOM++ via PyROOM++ serves as a proof-of-concept for the realization of the proposed language extensions. Furthermore, with the use of Python as an “action language” – Python is an object-oriented, dynamically typed language – we can show that rapid model prototyping is not only a vision but becomes a suitable approach.

Chapter 7: Methodology

Language extensions to ROOM and their realization in a modeling tool are one major concern of this work. Another major concern is the systematic approach in modeling complex communication systems. Chapter 7 wraps up the method we aimed for in this work and which has been manifested in the previous chapters. The methodology provides a detailed set of instructions to the hand of a modeler, which should support the modeler to convert a telecommunication standard, the results of a prestudy etc. into a model of the system architecture. The instructions are complemented by heuristics and modeling patterns and examples of how they can be expressed in ROOM++.

To demonstrate the methodology as a working method, the case study presented in the previous section of this introduction chapter is revisited. While SIGTRAN remains a complex subject of technology, our models uncover the basic structuring and the use of address spaces in SIGTRAN. Essentially, SIGTRAN gets much more clearer from our models than from the “models” presented in the standards.

Chapter 8: Related Work

In chapter 8 we relate our approach to other published work.

Chapter 9: Summary and Outlook

Chapter 9 closes this work with a summary and an outlook.

Chapter 2

Foundations

Before one starts, one needs to choose a framework, a context for reasoning and arguing about a topic, i.e. one has to take a viewpoint on the world, a way how to look at the domain of concern.

In section 2.1 we will take a closer look on two of such frameworks: the OSI Reference Model and the TCP/IP Reference Model. Both reference models are the dominant models of the domain: OSI shaped and still shapes the design and structure of telecommunication systems, TCP/IP laid the basis for the Internet technology. We will come up with a harmonized view on OSI and TCP/IP. A complementary view, presented in section 2.2, concerns the general structure of a real-time system. Any communication system is technically constructed out of a set of real-time systems. Section 2.3 is a brief tutorial on ROOM, the modeling language we will use in subsequent chapters to describe models and patterns of solution. To support a formalized reasoning on basic design principles, we will introduce a suitable mathematical formalism in section 2.4. Section 2.5 is a summary of key points we draw up in this chapter.

2.1 Reference Models: OSI and TCP/IP

As a prerequisite to discuss architecture modeling we need to establish some consensus on the view we have on the subject matter of communication systems. A *domain*, such as communication systems, can be thought of as a “world” with its own conceptual entities or objects [SM92]. A *Reference Model* (RM) (sometimes also called *reference framework*) pre-structures a domain; it proposes a set of conceptions and relations and thereby implies a specific view and understanding of the domain. The two dominating frameworks for communication systems are the Open Systems Interconnection (OSI) Reference Model and the TCP/IP Reference Model.

2.1.1 Historical Background

The Open Systems Interconnection Reference Model (OSI RM) [ITU94] has laid a solid foundation for understanding distributed open system intercommunication. It has been developed in the early 1980s, but became soon “knocked down” by another communication architecture developed, which is sometimes called the Internet architecture and usually designated by the Transmission Control Protocol (TCP) [Pos81b] and the Internet Protocol (IP) [Pos81a] suite. In practice, OSI was a commercial failure, it has never been fully implemented, only some few protocols of OSI have.¹ The OSI RM turned out to be by far too theoretic and unnecessarily complex; printed out, the whole OSI standard easily piles up to a paper staple of one meter. Outstanding expertise and a lot of time is needed just to digest the material. Seen from that point of view, it seems to be quite natural that the Internet architecture overrun OSI RM. TCP/IP, forming the backbone of the Internet architecture, is simple to understand, easy to read (it sums up to slightly more than one hundred pages), public available, and – most important – it was already implemented and bundled with Berkeley Unix in the early 1980s. At that point in time nobody could foresee the emergence of the Internet; but when it happened, TCP/IP conquered the world: the Internet revolution began on Unix machines, which already offered IP and TCP as an ideal environment for interconnecting computers. The “specify first and implement later” culture of ISO/ITU (the standardization bodies of OSI)² did not result in working code until today [PD00].

Although TCP/IP and the Internet architecture are well designed and a hot issue for everybody interested in internetworking, OSI RM was and still is the

¹Note that the abbreviation “OSI” refers to nearly the whole X-series of the ITU-T standard; sometimes “OSI” is used interchangeably with “OSI RM”. ITU-T stands for the telecommunication standardization sector of ITU.

²ISO stands for *International Organization for Standardization*, ITU for *International Telecommunication Union*.

main guiding principle in network design for telecommunication systems. Besides of its didactic value, OSI RM manifests some basic concepts and principles of open systems intercommunication, which are still in use and have influenced the design of today's communication systems, even TCP/IP RM. Compared to OSI, the TCP/IP RM does not provide such a complete and stringent conceptual reference framework. That is why the next section starts with discussing OSI RM; OSI RM precisely defines concepts and terminology. This eases entering the topic under discussion. Later on, we cover the TCP/IP RM. Some more information and critics about OSI RM and TCP/IP RM can be found elsewhere, for example in [PD00, Tan96, PC93].

2.1.2 The Architecture of OSI RM

The concept of layers is a key characteristic of all communication systems. It is a means to stepwise increase the degree of abstraction and to separate levels of abstraction by precisely defined interfaces, and is reflected by the use of protocol stacks. The OSI RM is the most prominent framework for a layered communication architecture. We do not repeat OSI RM to the full extent, we just would like to remind the reader of the basic outlook, see figure 2.1: Several network layers are stacked upon each other, each layer realizing a complete network of its own. Higher layer network services rely on lower layer services until a physical layer is reached.

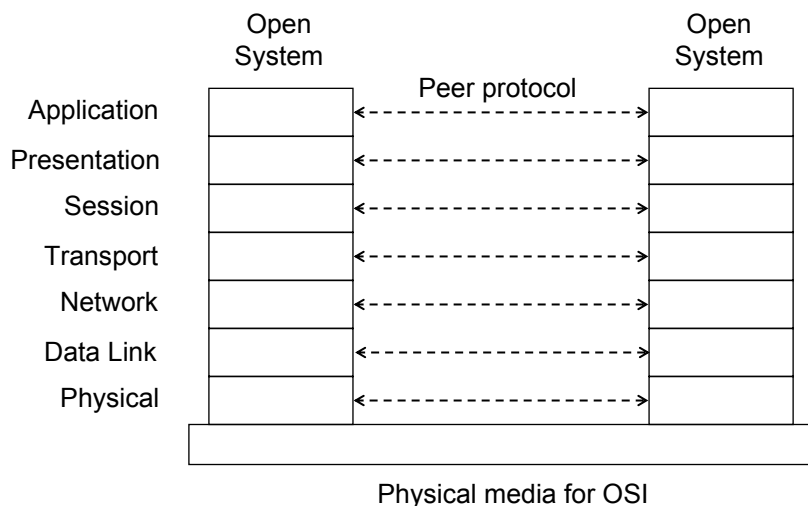


Figure 2.1: OSI seven layer reference model, see [ITU94, p.31]

Further introductory information can be either retrieved from the X-Series of the ITU-T recommendations or from textbooks. Almost any textbook on computer

networks and/or data communications gives an introduction into OSI RM, for example [Tan96, Hal96]. For the purpose of this discussion, we take the part of OSI RM that concerns the separation of one layer from another. This selection restricts the considerations on the cooperation of two arbitrarily selected but adjacent layers as shown in figure 2.2. The upper layer is referenced as layer (N), the lower layer as layer ($N - 1$).

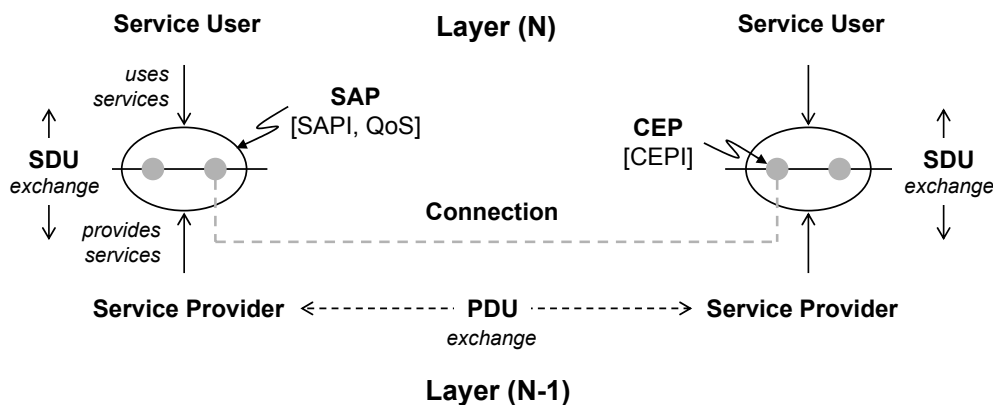


Figure 2.2: Layers, SAPs and CEPs according to OSI

The Layer Model in OSI RM

Any entity of layer (N) uses services provided by the adjacent lower layer ($N - 1$). Layer (N) is the *service user* that uses services provided by the *service provider*, namely layer ($N - 1$). By definition, *services* describe the capabilities of the provisioning layer and are expressed in terms of *service primitives*. Service primitives are a way of describing the service interface independent of an implementation; that is why they are also called *Abstract Service Primitives (ASP)*. ASPs could be implemented as procedure calls, library calls, signals, method calls etc., dependent on the implementation platform and language used. The “point” at which the service provider publishes its services for access to potential users is called *Service Access Point (SAP)*. The SAP defines the interface between a service user and a service provider. This not only includes a set of dedicated service primitives but also all allowed sequences of service primitives that are possibly exchanged between the service user and the service provider. An identifier, called *Service Access Point Identifier (SAPI)*, uniquely identifies the SAP within its namespace and is used for addressing purposes. For connection-oriented services, a *connection* describes a logical association between two peer entities located in different nodes. The SAP “owns” the *Connection Endpoints (CEP)* of the connection; each of them is identified by a *Connection Endpoint Identifier (CEPI)*.

Services not establishing a connection are called connectionless. *Quality of Service* (QoS) attributes may be attached to the SAP characterizing the qualities of a connection-oriented or connectionless communication relation in terms of reliability, throughput etc. A complete description of the basics of the OSI Reference Model with precise definitions of the concepts informally introduced is given in the standard [ITU94].

It is important to remember that OSI RM clearly distinguishes two communication relations: layer-to-layer (“vertical”) communication from peer-to-peer (“horizontal”) communication. “Vertical” communication refers to the exchange of information between layers (that is levels of abstraction usually within the same physical entity) in the form of *Service Data Units* (SDU). “Horizontal” communication refers to the exchange of information between remote peers. Remote peers are physically distributed and communicate with each other according to a protocol in the form of protocol messages, also called *Protocol Data Units* (PDU), thereby sharing the same level of protocol abstraction. PDUs are the vehicles for SDUs. A single SDU may be packaged into one or more PDUs. Such PDUs are also called *data* PDUs. Non-data PDUs are called *control* PDUs. In a multi-layer communication architecture, the service provisioning layer ($N - 1$) in figure 2.2 becomes the service user of the next lower layer ($N - 2$). The PDUs of layer ($N - 1$) get packaged into SDUs towards layer ($N - 2$). Thus, the concept of a connection is purely virtual unless a physical layer of transmission is reached. From a service user point of view the transfer of SDUs to/from its remote peer is transparent. Therefore, PDU handling of the service provisioning layer is not of interest. This is what is meant by the principle of abstraction in communication architectures.

The distinction vertical/horizontal is also reflected in terminology: layers communicate via an *interface*, remote peers via a *protocol*. This convention in terminology is seldomly followed and sometimes causes confusion between software and system engineers. Especially the term “protocol” (so is “peer-to-peer communication”) has a refined meaning in data and telecommunications. Even though the concept of a protocol is generally defined as a set of messages and rules,³ software engineers assume a reliable, indestructible communication relation in their software systems, whereas data/telecommunication engineers have to face the “real” world: they have to add error correction, connection control, flow control and so on as an integral part to the protocol. A communication relation between remote peers can always break, be subject to noise, congestion etc. This aspect, which characterizes distributed communication, is the reason why data and telecommunication engineers introduced layers of protocol and service abstraction into their

³Take e.g. the definitions given by [Tan96, p.27] as a representative of the data communication camp and [Bal96, p.191] as a representative of the software engineering camp.

models.

The most striking observation regarding layers in telecommunications is that the SAP and its “included” CEPs are the key concepts. OSI does not reveal how these concept can be noted or visualized, nor does it formally define the semantics of the SAP and the CEP concept. Early examples of capturing the SAP concept can be found in the standard of the Specification and Description Language (SDL) [ITU99a]. Since SDL was developed with a strong impetus of the telecommunication industry, the designers of SDL provided some application suggestions for typical telecommunication problems. The SAP was no exception. The suggestion that is given as an example for layering by service access “modeling” is described in [ITU93g] and elaborated on in [EHS97]. It is distinguished between a service user and a service provider but the service access is reduced to signal lists to specify the SAP interface. The CEP disappears completely. The example indicates that although the SAP and the CEP have been well-known concepts for quite some time now (more than 15 years) they do not seem to be sufficiently understood as modeling concepts, which requires some more attention and careful design. OSI does not give any precise answer how to model(!) systems according to the reference model. Take, for example, connections: typically, connections are dynamic and not static, they need to be set-up. What does that mean for the abstraction of a connection? Where does a connection come from? Out of the blue? What about connectionless communication, how is that modeled?

It is a characteristic of OSI RM and the subsequently presented TCP/IP RM to focus on vertical communication and to even subsume horizontal communication under vertical communication. That is why the CEP is regarded as something internal of the SAP, see figure 2.2; this bias towards vertical communication is one of the main (mental) hinders that prevent designers to clearly abstract communication systems on a specific layer of horizontal communication. OSI RM is simply not clear on that issue how to properly abstract communication layers.

A great deal of this work concerns the vertical and the horizontal interface and an approach that respects vertical and horizontal communication as equally important design principles.

Service Access Example

A simple example illustrates the service access and its functioning according to OSI. The example is based upon the “Abracadabra” service as given in [ITU93f, Annex F], which describes a reliable, connection-oriented service. The “Abracadabra” service is composed of the service primitives shown in following list:

- connection.request and connection.indication
- connection.response and connection.confirmation

- data.request and data.indication
- disconnect.request and disconnect.indication

The service primitives follow the format *primitive name* “dot” *primitive type*. The service primitive type is determined by the role of the service user and the service provider, respectively, and the direction of provisioning. When the service user acts as a requestor it may send a *request* to and receive an *confirmation* from the service provider. When, on the contrary, the service provider acts as an indicator it may send an *indication* to and receive a *response* from the service user. These four types cover all possible communication relations independent of the service primitive name. Usually, parameters are attached to service primitives. For example, a connection request has to include the sender’s and the receiver’s address, otherwise neither the request can be targeted towards a specific communication partner nor the response re-addressed to the source. For the sake of brevity parameters have been left out.

The interface by means of the SAP can be described by a Finite State Machine (FSM), see figure 2.3; the notation used is informal. The FSM specifies all allowed sequences of service primitives of the service incarnated by the SAP. In the symmetrical version of the example service any user can initiate a connection request, there is no client-server relationship. Supposed that we are having two service users A and B, and let A wish to establish a communication relation with B for data exchange. Since A cannot communicate directly to B without any foreign help, A sends a connection.request to its service provider. Assumed that both SAPs of A (SAP-A) and B (SAP-B) are in the initial state NULL, A’s request causes a transition from NULL to state Connection Pending of SAP-A. It is out of scope of this discussion, but somehow the peer service provider at B’s side delivers a connection.indication to B as a reaction on A’s request. SAP-B changes to state Connection Pending as well. B submits a connection.response to its service provider indicating that it accepts the connection request. SAP-B changes to state Data Transfer. In case that B would like to reject the request it alternatively submits a disconnect.indication, returning SAP-B to state NULL. Again, A’s service provider is “magically” informed by B’s service provider about B’s decision. If B agrees to the connection request, A’s service provider delivers a connection.confirmation to A. SAP-A follows changing to state Data Transfer. On a bi-directional connection any user may now submit a data.request, which results in delivery of a data.indication to the other party. At any time each of the users may want to actively close an established connection by sending a disconnect.request. Similarly, any user can be informed anytime via a disconnect.indication that the connection does not exist anymore.

Note that we exemplified vertical communication, i.e. service access in this subsection. We always need to be absolutely clear about which aspect of commu-

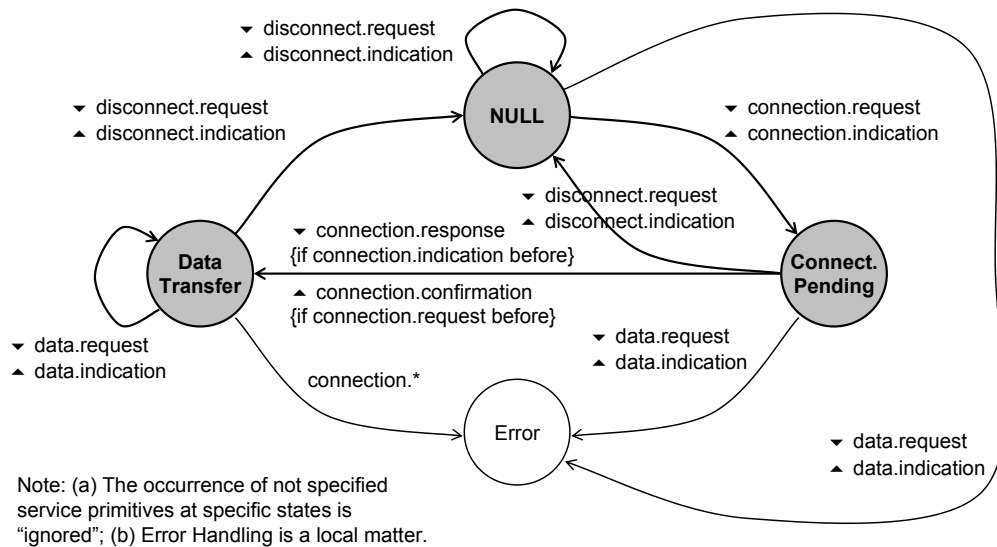


Figure 2.3: Finite state machine of the SAP

nication we are talking about in order to not confuse different sorts of relations: inter-layer communication via service access interfaces is not the same as intra-layer communication is between peers of a network layer. Horizontal communication via protocols will be discussed later. Horizontal communication is not made explicit in OSI RM and TCP/IP RM.

2.1.3 The Architecture of TCP/IP RM

The other well-known reference model for communication systems is the widely-spread Internet protocol suite, namely TCP/IP (Transmission Control Protocol/Internet Protocol). Because of the “implement first and (maybe) document later” mentality in the early days during the birth of the Internet, the reference model is nowhere explicitly written down. Nonetheless, TCP and IP are the dominating protocols used for *interconnecting networks* (the reason for the name “Internet”) so that they became *the* reference for the whole Internet communication technology. Actually, the Internet as we know it today consists of several dozen protocols, but the main spirit in design still honors the TCP/IP reference model.

The Layer Model in TCP/IP RM

The TCP/IP protocol suite is designed according to the same general principles as OSI is, both reference models have much in common. The difference is rather to be found in the terminology and the philosophy about the layers required. Figure 2.4 contrasts the protocol stack of OSI and the TCP/IP RM. The transport layer

is typically realized by TCP or UDP (User Datagram Protocol) [Pos80], the network layer by IP. UDP and IP are connectionless protocols, TCP is a connection-oriented protocol. Underneath IP a variety of technologies may implement the lower layer(s); the Ethernet [Spu00] is a standard example. Atop TCP comes the application layer; there was no need seen for a presentation and session layer in TCP/IP RM. A very popular application protocol of the Internet is the Hypertext Transfer Protocol (HTTP) [FGM⁺99], a protocol for browsing and surfing the World Wide Web (WWW). Other examples are the File Transfer Protocol (FTP) [PR85] and the Simple Mail Transfer Protocol (SMTP) [Kle01].

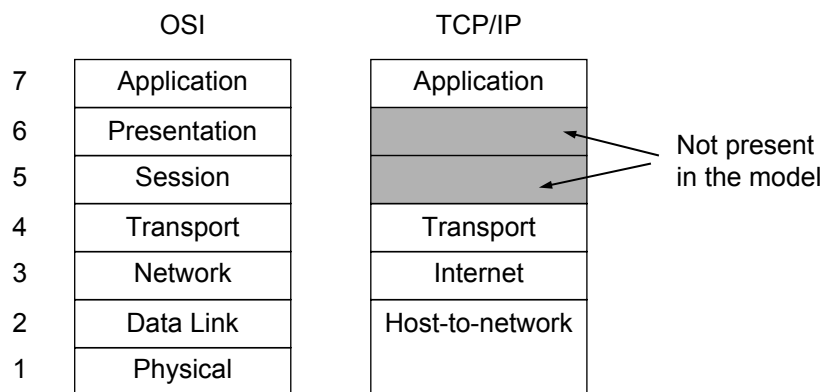


Figure 2.4: The TCP/IP reference model, taken from [Tan96, p.36].

Table 2.1 lists some of the key terms we have introduced in the OSI section above and maps them to the corresponding terms in TCP/IP. For example, *connection endpoint* corresponds to *socket* in TCP/IP, a *service access point* to *port*, and *service primitive* to *service call*. Throughout this work, we will mostly use the OSI terminology instead of the TCP/IP terminology. Reason is that the OSI terminology is much better defined than TCP/IP.

Table 2.1: Mapping of OSI terminology to TCP/IP terminology

OSI Term	TCP/IP Term
service provider	host
service access point	port
service primitive	(service) call
connection	connection
connection endpoint	socket
connection endpoint identifier	socket address

Service Access Example

The TCP layer provides some few primitives to the application layer. The TCP protocol is specified in [Pos81b], table 2.2 lists the primitives provided.⁴ Note, that the service primitives do not follow the same strict classification in primitive types as OSI does; there are no explicit service requests, confirmations etc. TCP/IP service primitives are pragmatic in design, since they can be interpreted as library or procedure calls of call-by-value type with possible return values. Remember that TCP/IP was first implemented on UNIX machines. In that sense, the TCP/IP specification is less “abstract” than OSI is.

Table 2.2: Service primitives of TCP as specified in [Pos81b]; “m” indicates mandatory, “o” optional parameters

Primitive	Parameters	Return Value
LISTEN	(shortcut for “passive” OPEN)	
OPEN	m local port, foreign socket, active/passive o timeout, precedence, security/compartments, options	local connection name
SEND	m local connection name, buffer address, byte count, push flag, urgent flag o timeout	
RECEIVE	m local connection name, buffer address, byte count	byte count, urgent flag, push flag
CLOSE	m connection name	
STATUS	m local connection name	status data
ABORT	m local connection name	

When the primitive OPEN is submitted, TCP establishes a connection between the caller and the callee specified in the parameters of OPEN. Correspondingly, CLOSE causes the connection to be released. SEND requests the data given to be sent over the specified connection. By submitting RECEIVE, the TCP user indicates readiness to receive data. The ABORT primitive requests an “urgent” release of the connection accepting even pending data not to be processed anymore. STATUS is an implementation dependent user command, optional in use, and returns information about the local connection. Some implementations also use the

⁴Clarifications and bug fixes of TCP are detailed in [Bra89], extensions are given in [JBB92].

LISTEN primitive, which blocks the service user, meaning that the service user indicates to the provider that it is ready to accept any call from the provider. LISTEN is identical to a “passive” OPEN. That is, the service user is ready to react e.g. on interrupts triggered by the TCP layer. Similarly, RECEIVE indicates the user’s willingness to process any data received by the TCP layer. The primitives given in table 2.2 (except LISTEN; STATUS is optional) form the minimum set of services all TCP implementations have to support. The transfer of SDUs between the application and the TCP layer is done via buffers.

The state machine, which describes the behavior of the SAP between TCP and the service user is quite simple and shows quite much similarities to the *abracadabra* service specification in the section above. That is why a corresponding state machine is not presented here.

Two small Python [Lut01] programs taken from the Python documentation⁵ exemplify the use of a TCP service provider to establish a connection, transfer data, and eventually release the connection. Figure 2.5 shows the code of a very simple server application relying on TCP as a transport network layer, and figure 2.6 the corresponding code of a rudimentary client application on top of TCP.

```
# Echo server program
import socket

HOST = ''                # Symbolic name meaning the local host
PORT = 50007            # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

Figure 2.5: A simple server using TCP services written in Python

The Python module used for socket programming (line #2 in both listings) provides access to the commonly used Berkeley Standard Distribution (BSD) socket interface [Ste98]. What is intended to show here is to give the reader an impression of how surprisingly simple it is to access TCP services. Since the programs are almost self-explaining, only some few comments are given: Before the client contacts the server, both programs have to instantiate a socket object *s*; the server

⁵See also <http://www.python.org>; the programs are written in Python version 2.2.

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'      # The remote host
PORT = 50007                # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', 'data'
```

Figure 2.6: A simple client using TCP services written in Python

has to be executed prior to the client, of course. The server's bind command associates the socket object to a socket address, which is composed of the IP address of the server's host and the port address. After that the service goes to state LISTEN, waiting for a connection request. If a client connects, the server waits for data transmission over the connection (conn.recv with a buffer size of 1024 bytes) and echoes the data back to the client before it closes the connection. The client program requests a connection to the server's socket address and sends the "Hallo, world" over the connection. Data received by the client is printed before the connection is closed.

2.1.4 Conclusions

The OSI and the TCP/IP reference model have much in common. They differ more in terminology and implementation of concrete protocol stacks than in their general design principle. Decisive is the stacking of complete network layers, and the notion of services and protocols. The following few conceptions are central for both reference models:

- Horizontal or intra-layer communication goes via *connection endpoints*; the communication relation may be *connectionless* or *connection-oriented*.
- The communication relation is specified in form of a (*communication*) *protocol*. Sometimes we will refer to communication protocol messages as Protocol Data Units (PDUs).
- Vertical or inter-layer communication goes via (*service*) *interfaces*, also called *service access points*.

- The service relation is specified in form of a *service protocol*, often also by means of *service primitives*. We will also refer to individual service protocol messages as Service Data Units (SDUs).

Please note that this list results from an unbiased, fresh few on OSI RM and TCP/IP RM. So far, in literature and even in the OSI standard, communication systems are notoriously viewed from an implementation point of view. With an implementation focus, it is the service interface and the communication protocol that can be put in code; horizontal communication relations remain purely virtual and are, at best, hidden in some data structures. This implementation perspective is also apparent in the OSI RM, the concept of a (virtual) connection remains a helping construct. Very often it is overseen that the U-shape of the OSI RM (see figure 2.1) is caused by this implementation perspective that requires to implement service interfaces down to the lowest, the physical layer. Even a respected author like TANENBAUM shows to be biased. He regards three concepts as central to the OSI model: services, interfaces, and protocols [Tan03, p.44] – and forgets about the forth one, the connection endpoint as the corresponding interface for protocols.

This strong bias towards an implementation perspective is – according to the author’s opinion – the main reason why there has not been developed a systematic (unbiased) modeling approach for communication systems yet. It is a major contribution of this work to re-discover and re-install basic structuring principles of communication systems. As the attentive reader might have noticed, the symmetry of the concept pairs connection endpoint/communication protocol and service access point/service protocol is a first step in this direction. There is no precedence of one concept pair over the other.

Consequently, we cover each concept pair including all related implications in main chapters. Horizontal communication with its core concepts of connection endpoints and communication protocols cover the aspect of *distribution* in a communication system. We will devote a complete chapter, chapter 4, on that subject. Vertical communication with its core concepts of service access points and service protocols cover the aspect of *layering* networks. Another chapter, chapter 5, is devoted to that subject. As we will see, distribution and layering are much more general design principles than horizontal and vertical communication are.

To summarize: Our approach originates from a harmonized view on OSI RM and TCP/IP, which have very much in common on their general design principles. We just break with a long lasting tradition to favor an implementation perspective. As a result of that we span our modeling space along the principles of

- distribution and
- layering.

In the next section, we will add another, complementary view on communication systems: networked real-time systems. This view will contribute a third major principle.

2.2 From (Embedded) Real-Time Systems to Communication Networks

A communication network is composed of distributed but communicating nodes of resources, each of them controlled by an embedded real-time system. In this section, the properties and the general structure of a real-time system and its extension to a communication network are discussed. In specific, we develop an understanding of the types of communication as a result of an controller/controllee relationship and conclude basic language conceptions for building real-time systems.

2.2.1 (Embedded) Real-Time Systems

The use and the understanding of the term “real-time system” is not consistent in the literature. It is a mixture of characterizing attributes and structural properties of a system. For example: On one hand it is said that a real-time system fulfills timing constraints, i.e. a real-time system has to react to a stimulus in a certain time frame; in this example the guaranteed response time is an attribute, which characterizes a real-time system. On the other hand, real-time systems are often classified as “embedded systems”. An embedded system is seen as a specific part of a larger system, which is a structural aspect. Besides this lack of clarity in terminology, there is not even common agreement on the word “real-time”. The following paragraphs summarize findings from studying the literature.

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time in which the results are produced [Sta88]. After more than a decade this definition still seems to be the greatest common denominator. Here, “real-time” is an attribute to “system”. Because of their specific field of application, further attributes are usually associated with real-time systems. Included in this category are e.g. reliability, fault tolerance, adaptability, and speed [Tuc97].

The most popular classification is the distinction in *hard* and *soft* real-time systems. Hard real-time systems are under deadline constraints. Passing a deadline is considered unacceptable. A soft real-time system retains some tasks, which are still valuable for execution even if they miss their deadlines [Tuc97]. Following [SGW94] telephony systems belong to the class of soft real-time systems: passing of deadlines is accepted as long as the number of failures is below a defined threshold. While this might be true in general there are other components in telecommunication networks, which have to fulfill hard real-time constraints. For example, the time delay perceived as acceptable for voice transmission in a speech conversation places tough time limitations on a mobile phone for speech

en- and decoding including cyphering and channel coding.

A very rudimentary structure of the basic elements of a real-time system is given by [SGW94], see figure 2.7: it consists of hardware, *sensors* and *effectors*, the environment, and software. The sensors and effectors interact with the environment, the software controls the actions of the hardware via a hardware interface. A similar description using different terminology can be found in [Sta96]: A real-time system consists of a controlling and a controlled system. The controlling system interacts with its environment using information about the environment available from various sensors and activating elements in the environment through “actuators”. The controlled system can be viewed as the environment with which the computer interacts.

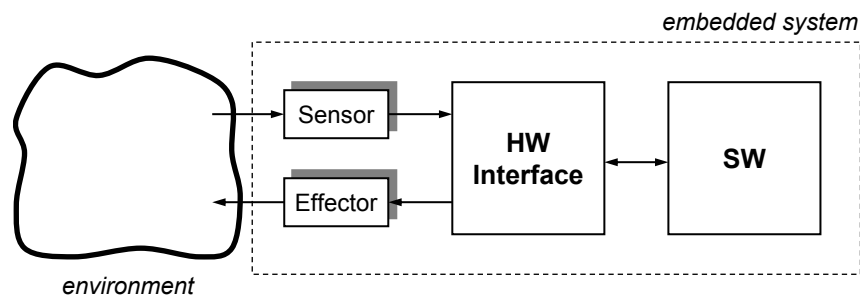


Figure 2.7: The basic elements of a real-time system

There is a loose reasoning as to why timing aspects and structural issues of a real-time system are related: Timing correctness requirements arise because of the physical impact of the controlling systems’ activities upon its environment. That means that the environment needs to be monitored periodically as well as that sensed information needs to be processed in time [Tuc97]. This implies that we have to distinguish the environment from a controlling part and that detecting and acting devices are needed.

The definition of an embedded system is vague; it mainly describes a structural aspect. In its most general form, an embedded system is simply a computer system hidden in a technical product [Sim99]. A more concrete definition is, most embedded systems consist of a small microcontroller and limited software situated within e.g. an automobile or a video recorder [S⁺96]. Three issues seem to be important here: (1) size matters, (2) an embedded system is part of a technical system, and (3) it serves the purpose of the technical system and not vice versa. Issue (3) especially helps delimitate non-embedded systems from embedded systems. A PC (Personal Computer) for instance is a general purpose computing machine, the software and the CPU (Central Processing Unit) are an integral part of it. This eliminates a PC from being an embedded system. A counterexample

might be a mobile phone. The DSP (Digital Signal Processing) chip and its software serve a single purpose: to offer phone functionality.

Embedded systems may or may not have real-time constraints [Sta96], but many real-time systems are embedded systems [S⁺96].

To summarize: The special character of systems, which have a physical impact on the “real” world by means of reactivity is most significantly described by the requirement on the timing constraints to be met by the system. Such systems are called *real-time systems*. Additional properties, which reflect other aspects of the physical impact character, include reliability, fault tolerance, stability, safety and so on. As yet there is no commonly agreed list of properties, which constitute a real-time system. Moreover, the physical impact nature of such systems implies a rough structure (Fig. 2.7): a controlling part interacting with the environment (the controlled part) through sensors and effectors. The hardware mediates between the sensors/effectors and the software of the system. Many real-time systems are embedded systems, that means they serve a specific purpose in a technical system. This is actually the case for all nodes in a telecommunication system.

The model identified so far is not specific enough to allow any detailed insight into the nature of real-time systems, especially for structural properties. There is a need for further considerations. A slightly extended model of real-time systems will help improve the understanding of such systems from an architectural and modeling point of view.

2.2.2 An Extended Model of Real-Time Systems

Figure 2.8 shows a slightly extended model for real-time systems. What adds some more structural aspects to the model are the distinction of a technical system and the environment, the explicit mentioning of a user, and the intrinsic motivated association of the software of the controlling system with the controlled system by means of a Controlled Domain Model (CDM). As the reader will notice, the following discussion of the extensions is influenced by a system theory approach. After the discussion we will draw first conclusions about suitable modeling conceptions.

The Model

The basic elements of a real-time model still remain valid: it consists of a controlled system and a controlling system. Sensors and effectors classify two types of devices, which reflect relations of impact. Sensors communicate events of the controlled system to the controlling system, whereas effectors do the opposite. The model tries to separate structural issues from system attributes. For example,

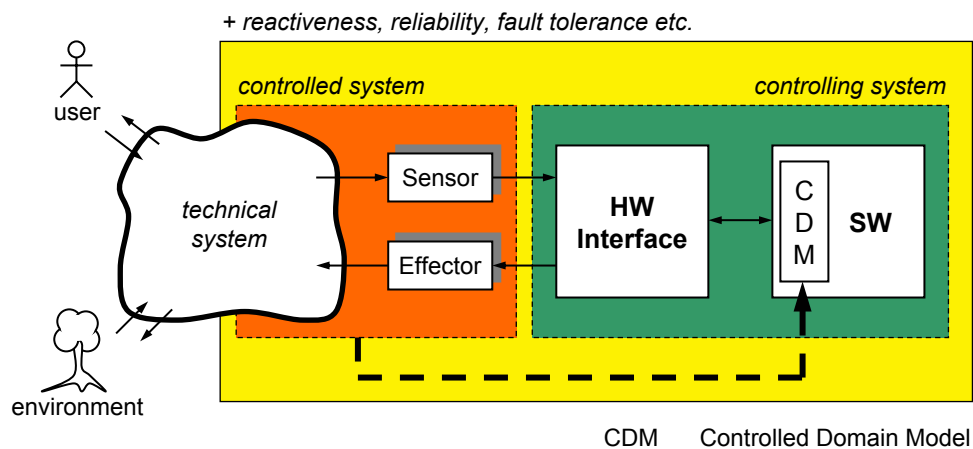


Figure 2.8: An extended model for real-time systems

the decomposition of a system in a controlled and a controlling system is a structural property. Reactiveness under specific time constraints is an attribute, which leads to the classification *real-time*. As previously mentioned, further properties might refine the attribute “real-time”. This is indicated in figure 2.8 by some “attributing” text.

The box labeled with “HW” condenses the hardware required to establish an interface between the sensors/actuators and the software; it could be, for example, an analogue/digital (A/D) converter. The computing platform as well as other hardware devices are neglected but can be subsumed under the “HW” box if needs be. Note that we cannot make any assumptions about the hardware structure. We do not know if a pair of sensors/actuators is associated with a single A/D converter or not.

Sensors and actuators could be interpreted as exporting, respectively importing reference points in the controlled system. Consequently, sensors and actuators belong to the controlled system. This fact is not sufficiently reflected in figure 2.8. Reference points are associated with an entity in the controlled system. An *exporting* reference point is a defined exit point of matter, information, or energy from the technical system to the external world. The intention is to allow the state of the external world to be influenced in a specific way. An *importing reference* point is a defined entry point of matter, information, or energy from the external world to the technical system, which allows the state of the technical system to be changed. Note that reference points are an intellectual concept and denote only those entry/exit points, which are relevant to the technical system, others are neglected.

Physically, a transfer of matter, information, or energy is always equivalent to a change of state; it is experienced as a loss or gain of matter, information, or energy. Even though every technical system is physical, the point is that a technical

system is a construction that is sensitive to some physical changes in state at its reference points and while remaining relatively insensitive to others. A sensor, for example, ideally is a technically state insensitive reference point; it should transfer information about the technical system's internals without changing a technically sensitive state. In this sense, a sensor simply observes the controlled system and reports its observations to the controlling system. In contrast, an effector should have impact on a technically sensitive state.

The controlled system is not equal to the environment, it covers just a part of it as there are only technical entities with associated reference points (sensors and effectors). This view is fundamentally different to the traditional one of real-time systems. What is new is the introduction of a *technical system*. The technical system is composed of technical entities. Many of these entities have an impact on or are in connection with the physical world, namely the environment. The relation between technical entities is defined by the technical system. These relations might be mechanical, electromagnetic, chemical, etc. Additionally, the technical system might interact with a user. A system, which has no user interface, is called *autonomous*. The interaction between the technical system and its environment or a user is also via reference points in the technical system.

In practice, a real-time system is usually seen as a component of a technical system. For example, it would be inconceivable for today's automobiles not to have computers controlling the fuel-injection engine. According to our view, the controlling system resides outside the technical system because reference points mark interfaces to the external world and hide implementation details. Interfaces are some kind of ports to the outside and reflect "expectations" of the external world; it is insignificant as to how the external world looks. For example: The component "engine" of a technical system "car" demands specific synchronizing conditions on fuel-injection, which are formulated as requirements. Maybe the engineers come up with a mechanical solution or they decide to define suitable reference points for sensors and effectors and control fuel-injection by a computing device.⁶ Once a specific solution is established, it is typically regarded as a technical component of the system. This ultimately blurs the system's borders, which conceptionally still exist. Here, we will stick to a clearer conceptional view.

To wrap up: (1) The controlling system, users, and the environment are external to the technical system; the distinction made is a classification of the technical system's external world. (2) Reference points are associated with technical entities and point out interfaces to the external world. (3) Reference points are either

⁶In fact, the topic is a bit tricky: a mechanical solution requires different kinds of reference points compared to a computer controlled solution. Ergo, the definition of reference points marks a decision that is a constraint on the solution space. The point is that reference points for a real-time system shift domain! Sensors and effectors mediate between the states of e.g. a mechanical or chemical domain and the software domain of a controlling system.

importing or exporting. Exporting reference points can be sensitive or insensitive to the technical state of the technical system.⁷

Every transactional system that aims to influence its environment has to be aware of the current state of the environment [Ber99].⁸ However, a pure representation of states does not help unless one knows how the states are related and influence each other. In other words, a domain model is needed, or – according to [Ber99] – a so-called Process Domain Model (PDM). A PDM does not only define the static structure of a domain (a so-called Structural Domain Model, SDM), it additionally takes into consideration the processing dimension. For better naming conformity, we prefer the name Controlled Domain Model (CDM) instead of PDM.

It is important to see that one only needs to have a CDM of the controlled system and not of the whole technical system. This is also new and in contrast to traditional views on real-time systems. For the purpose of the real-time system, there is no need to completely understand the technical system, its entities and their relations; however, it is essential to understand the function of the controlled system. The only thing that must be understood by the real-time system is, how the entities of the controlled system and the sensors and effectors function and possibly relate to each other. In other words, the software in the controlling system must have a built in accurate representation (model) of the controlled system. Formally speaking, there exists an association between the software of the controlling system and the controlled system by means of a CDM.

The interesting consequence is that we have an indication of a structural property of the software: it must somehow reflect the domain it is controlling. The question is *how* and to which extend the CDM structures the software. In some cases, formulas might model the controlled system, in others a set of context rules might provide an alternative. More complicated cases require data structures and models, which reflect properties and relations of the controlled domain. We can state the hypothesis that the more complex the controlled system and the task of controlling it is, the more evident the relation of the software internals (data structures, software architecture) and the controlled system structure (the CDM) must be. In an extreme scenario the software mirrors the controlled system structure in a one-to-one manner. In any case, the software specifies a model of the controlled system.

⁷A state insensitive importing reference point is of no value; according to the definition, a reference point is of relevance to the technical system. Ergo, state insensitive importing reference points do not exist.

⁸A transactional computation depends on the state of the environment in which it is carried out [Ber99].

Conclusions

Given all this, how can we abstract the elements of a real-time system such that we get a notion of language constructs needed for modeling a real-time system?

As we learned, a key characteristic of the controlling system is that it behaves reactively i.e. on a stimulus it reacts with a response; so does – just in an inverse sense – the controlled system. Both, the controlling and the controlled system have a state and behavior. The notion of an *(re)active object* having its own thread of control is a meaningful abstraction for both systems. The spontaneous rise of one or more stimuli or responses at the very same point in time gives a preference for an *asynchronous communication paradigm*.

We also learned that the sensors/effectors together with the HW can be regarded as defined *interfaces* that transform means of notification from one media towards another media (here from a physical world of e.g. mechanical devices to units of software). An appropriate abstraction of a notification mechanism that initiates an activity at the receiving (re)active object is the notion of a *message*. Messages support the demand for asynchronous communication. With the notions of (re)active objects, interfaces, and messages, the distinction between hardware and software has become obsolete.

The controlling system has an internal representation of the state space of the controlled system (the CDM) in order to meaningfully operate on the controlled system. It is the CDM that qualifies a distinction of roles: a controlled system and a controlling system. The controlled system does not have an internal state representation of the controlling system. For convenience, we also call the controlled system the *resource provider*, the controlling system the *resource user*, and the CDM the user's *resource model*. We will use these terms later on in a more generic context that is not bound specifically to the notion of real-time systems.

So far, we are not very precise about what we actually mean by *(re)active object*, *message* and so on, although we use common terms of computer science. The meaning will be more refined, when we come to a realization of the mentioned conceptions in a concrete language, namely ROOM.

2.2.3 Collaborating Distributed Real-Time Systems

In particular, telecommunication systems are characterized by a network of distributed collaborating units. Therefore it is of special interest to discuss some consequences if two or more real-time systems modeled according to figure 2.9 collaborate and communicate to each other. In figure 2.9 some elements of a real-time system model are removed for reasons of simplicity.

The interaction of two technical systems is at first sight outside the context of the controlling system. Two technical systems can physically only interact via

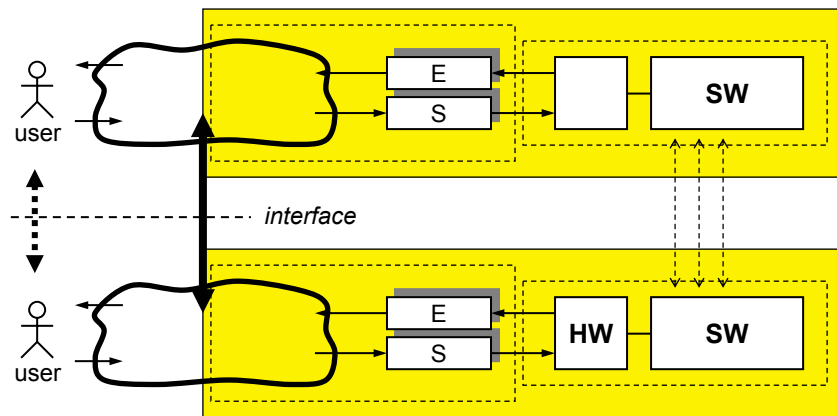


Figure 2.9: Collaborating distributed real-time systems

two or more technical entities. In principle, this can be any entity in the upper or lower technical system in figure 2.9. Communication is established via an external connecting entity. This entity is usually part of the environment. Two phones could be, for example, either connected by a fiber optics cable or, in the case of mobile phones, by the “air” (electromagnetic waves). Instead of an external entity, a shared technical entity might also be an option, but this blurs the system’s borders. To be conceptionally clear, such a shared resource should be moved out and be defined as external, while replacement entities in the technical systems have to be introduced if required.

If some sort of connectivity of the technical system is of relevance for the controlled system, the connectivity is reflected in the resource model of the controlling system as well. That means that the communication relation between two (or more) technical systems has its correspondence between resources of the resource model of the controlling system; this is indicated by the dashed arrow next to the solid arrow in figure 2.9. If the resource model is internally structured in hierarchical layers of abstraction (as it usually is), we will also find the communication relation abstracted on these layers. This abstraction technique is well known in the telecommunication domain; the most prominent example is the OSI (Open Systems Interconnection) reference model [ITU94]. The notion of layered communication relations is shown by further dashed arrows between the controlling systems in figure 2.9. If communication facilities are offered as services to users, the users experience a high-level of communication (as a result of abstraction levels in the controlling system) that goes far beyond the facilities of the communication relation between entities of the technical system. If the real-time system is autonomous, it is an application inside the controlling system that benefits from a high-level communication instead of an “external” user.

2.2.4 Collaborating Networks

Very often it is overseen that not only nodes may compose a network but also networks may collaborate and compose more complex networks. This sort of composition is shown in figure 2.10. The figure is not 100% accurate, since the networks must either share parts of their technical system or establish a communication relationship in order to cooperate.

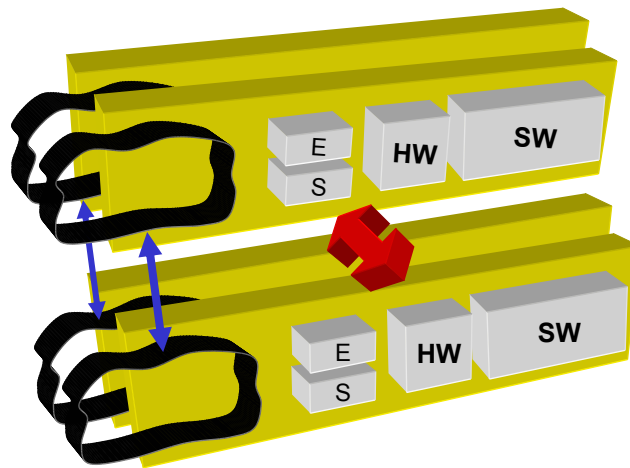


Figure 2.10: Collaborating networks

The individual networks of the “supernetwork” are called *planes*. We already gave an example and an explanation in the introduction chapter: very common is the distinction in a control plane and a user plane. It is a repetition of the controller/controllee relationship just on a network level.

2.2.5 Summary

The discussion of real-time systems aimed at two points: (1) We motivated basic language conceptions required to model real-time systems. A modeling language that suits the discussed needs for real-time systems is presented in the next section. (2) We discussed the consequences of a controller/controllee relationship: that the controller must have an internal representation of the controllee at his disposal, otherwise the controller cannot use the controllee for his own purposes. We agreed upon to speak more generally of resources, resource models, resource users and resource providers. When we moved on to discuss distributed real-time systems, we saw that distribution demands communication relations for collaboration and that these communication relations are reflected in the resource model of the resource user. We also learned that communication relations are usually

abstracted to more high-level communication relations. Finally, we also saw that planes of networks build up supernetworks.

It makes sense to contrast a controller/controllee (user/provider) relationship against a peer-to-peer relationship. This distinction marks the third major design principle for our modeling approach. Decisive for the distinction is: Who has a Controlled Domain Model (CDM) of whom? In user/provider communication, the user has a resource model of the provider; in peer-to-peer communication, either both parties or no party has a resource model of the opposite side. This distinction is the basis for differentiating different (atomic) types of communication. We will devote a complete chapter, chapter 3, on that subject. The modeling space is now spanned along the principles of

- distribution,
- layering and
- types of communication.

What we do not consider are other issues of relevance that may shape the architecture of a communication system. Among others we exclude non-functional requirements such as reliability, maintainability, safety, security, compatibility, robustness, fault tolerance, and so on.

2.3 A Brief Primer on ROOM

The ROOM (Real-Time Object-Oriented Modeling) language plays an important role in this work: It is our preferred language for modeling complex communication systems. We use ROOM to describe models but also to describe patterns of solutions. From the insights we gain throughout this work, we will also derive extensions to ROOM that help catch specific conceptions in a much better way.

This section is a brief primer on ROOM. It does not function as a replacement of the authoritative ROOM introduction, the ROOM book, see [SGW94]. Rather we informally recap the most important features of ROOM. The intent is to provide readers with basic skills to understand the ROOM diagrams used in this work. A more formal treatment of ROOM is presented in chapter 6.

2.3.1 Structural Elements

Actor, Port, Message, Protocol

The ROOM language is built upon the notion of an *actor*.⁹ An actor represents a physical device or a software unit; it is a sort of active object that clearly separates its internals from the environment. Everything inside the actor, meaning the actor's structure and behavior, is not visible to the environment. Only at distinct points of interaction, so-called *ports*, the actor interfaces the environment. A port is somewhat comparable to an interface as known e.g. in the UML but the comparison blurs two important facts: (1) Ports in ROOM are not method interfaces but message interfaces. A *message* consists of a message name, priority and data. Messages may be incoming and/or outgoing at a port (the direction is always defined from the viewpoint of the actor). So, ports are message exchange points between the actor and its environment. (2) A port is not only an interface that tells the environment how to use the actor; it is also a definition of the actor's expectations on the environment. Therefore, a *protocol* is always associated with a port, which defines the set of incoming and outgoing messages that may pass the port.¹⁰ This symmetry of a port, that it specifies not only how the environment can use the actor but also how the actor can use the environment, is an important distinction to the notion of an interface in OO.

An actor is specified by means of an *actor class*. It may contain port specifications and further aspects we will come to in a minute. An actor class is symbolized

⁹Other terms and synonyms for actor are *component* and *capsule*.

¹⁰ROOM's definition of a protocol does not match the OSI definition: the former refers to a protocol as a set of messages, the latter includes message sequences as rules. We therefore prefer to call a defined set of messages a *message schema*, and a message schema plus rules a *protocol*. Nonetheless, in the ROOM context we will often stick to the term *protocol* in order to stay consistent with the terminology used in the ROOM book.

by a rectangular box with a thick black border. A port is figured by a small squared box that appears on the border of an actor class symbol. An example is shown in figure 2.11. By convention, actor class names start with a capital letter, port names with a small letter.

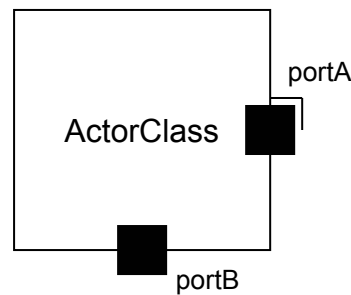


Figure 2.11: Actor class with ports

The message protocol (better: message schema) associated with a port is usually specified in text and does not explicitly appear in the diagram. What we can also see from figure 2.11 is that ports may have a *replication factor*, indicated by the “shadow line” at port portA. The precise number of replica of a port is not included in the graphical representation. Replication in fact is a way of indexing ports with e.g. portA[0] being the first port, portA[1] being the second one, and so on. Non-replicated ports have no index and appear just as e.g. portB.

Actor Reference, Binding, Contract

An actor can be composed of other actors. In ROOM, references describe compositions. That means, an actor class specification may reference zero or more other actor class specifications. Such a reference is called *actor reference*; it is a way to include other actors into the name space and life-time context of an actor. Circular references (A references B, and B references A) are forbidden. Per actor reference, a *replication factor* determines the maximum number of valid actors of the referenced actor class that can be put in context. By default, the replication factor is set to one.

The following types of references can be distinguished: an actor reference may be fixed, optional, imported or substitutable. These types specify run-time relations. For a *fixed* actor reference, actors of the referenced actor class are incarnated along with the incarnation of the composing actor. If the actor reference is declared as *optional*, actors of the referenced actor class can be dynamically created and destroyed during the life-time of the composing actor. The maximum number of allowed actors (given by the replication factor of the actor reference)

may not be exceeded. If declared as *imported*, an actor that already exists in another context of another composing actor is plugged-in at incarnation of the composing actor. That means, a single actor instance may act in two or more contexts of a composing actor: in the context of the “original” composing actor that created the actor and owns the permission to destroy it, and in the context of one or more other composing actors which imported that specific actor.¹¹ Imported actor references are a powerful tool to define different roles for different contexts of an actor and thereby to define patterns of collaboration. We will extensively make use of this feature. The risk is to misuse this feature and violate a design rule such as the “Law of Demeter” (only talk to your immediate friends) [HT99]. A *substitutable* actor reference means that any actor instance of that actor reference can be replaced by another actor, provided that the other actor’s class specification is compatible with the referenced actor class of the actor reference. Here, compatibility means that the other class specification supports at least the same set of ports (with the same message schema).

To build up complete structures of actor references, there have to be some means to interconnect their ports. This is done by so-called *bindings*, sometimes also referred to as *connectors*. A binding connects a port of an actor reference either with the port of another actor reference or with a port of the composing actor class. Bindings define communication relationships on class level. The auxiliary concept of a *contract* consists of a binding and the two interface components (ports) that the binding connects.

An example of an actor class specification that encompasses all the discussed modifications of an actor reference is shown in figure 2.12. Actor references are symbolized by a rectangular box with a thinner black border and can only appear “inside” the context (also called *decomposition frame*) of an actor class specification.¹² Names for actor references begin with a small letter. Names for bindings also begin with a small letter by convention. Sometimes, to avoid visual clutter, the names of bindings and ports are not displayed in the diagram. The replication factor of a replicated actor reference is displayed inside a box in the upper right-hand corner. Optionality is indicated by stripes. If imported, the actor reference is colored grey. Substitutability is indicated by a “+” symbol in the upper left-hand corner.

For obvious reasons, the message schemata of two ports connected by a binding must match. All or a subset of the messages outgoing one port must be specified as incoming for the other port and vice versa. To simplify matters, in ROOM both ports refer to the very same protocol specification. The port visualized by

¹¹Imported actor references cannot be combined with the optional reference modifier.

¹²Note that for practical reasons we sometimes do not draw actor classes and actor references with different lines of thickness. It is impossible to confuse actor classes with actor references, since the latter have to be drawn inside the former.

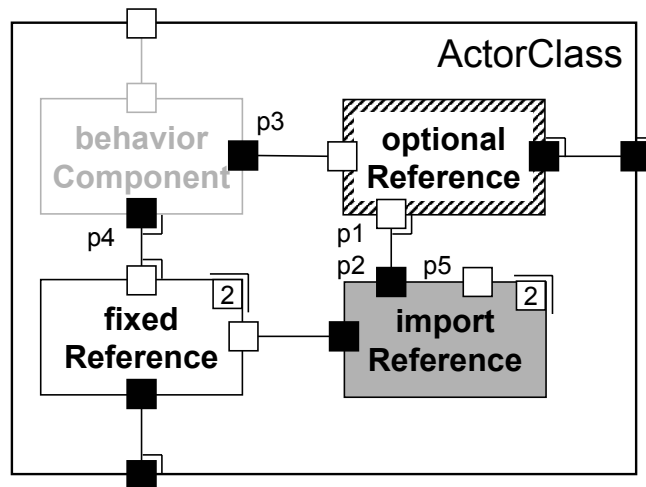


Figure 2.12: Actor class containing all types of actor references

a small black box takes the protocol specification as is; the opposite peer port (visualized by a small “white” box) “converts” the protocol specification, i.e. all messages defined as outgoing by the protocol are regarded as incoming for the opposite port, and all messages defined as incoming by the protocol are regarded as outgoing. The opposite peer port is said to *conjugate* the protocol. If the protocol is P , the conjugated version is noted as P^* .

Let us draw some special attention to contracts. All ROOM diagrams are on specification level (or class level, if you prefer to say) that are incarnated at some point in time during model execution. It is relatively easy to specify contracts (remember, a contract is a binding including its ports) that cannot be incarnated – which is a specification error and not a run-time error. The rule is to specify a contract such that potentially all port incarnations of one side of the contract have a unique port incarnation at the other side of the contract they can communicate to. That means, the maximum number of possible port incarnations at both sides of a binding specification have to match; otherwise, there are principally port instances left with an unspecified communication partner. To be concrete, let us have a look at figure 2.12. Port $p1$ has a replication factor of two; its owning actor reference has a replication factor of one. Therefore, two incarnations of $p1$ can be maximally created at run-time. The opposite peer port of $p1$, $p2$, has a replication factor of one; its owning actor reference a replication factor of two. The equation resolves to $2 = 2$, which verifies a valid contract specification. It takes some time to get used to see the consequences of replication structures on incarnation level. They are discussed in more detail in the ROOM book, see [SGW94, p.183ff.].

New in figure 2.12 is the component that specifies the actor class’ behavior. In fact, the behavior component is invisible; the behavior component’s border is

colored in grey just for demonstration purposes. Thus, all ports of an actor class specification that are not connected somewhere else are actually connected to the actor's behavior component; they are called *end ports*. Otherwise, they are called *relay ports*. All other ports (p3, p4) that “hang around” are also implicitly connected to the behavior component. *Reference ports* (the name for ports of actor references) that are not involved in a contract are actually not in use, see p5.

Note that in ROOM, relay and end ports have special notations. In our diagrams we deviate from ROOM in that we use black and white boxes for all kinds of ports. This notational simplification eases the drawing of ROOM diagrams but does not suffer semantic preciseness. Still it is unambiguous, which port is an end port, which one a relay port, and which one a reference port.

Layer Connection, Service Provision Point, Service Access Point

The notion of layers is a built-in concept in ROOM. Layering is a form of abstraction that is used to define “islands” of self-contained functionality that provide services to another “island” of functionality. In contrast to the *horizontal* structure of peer-to-peer communication between ports, layers represent a *vertical* organization of a system. The terms “horizontal” and “vertical” are apparently vague and indicate the difficulty for giving a precise definition of layers. Actually, the sort of interfaces used to describe layers are very close to ports. Despite the fact that ROOM is not absolutely clear on the layering concept, here we just present layering as is without further criticism. The layering concept will undergo a revision in later chapters.

The interface of an actor that provides (layer) *services* towards another actor is called *Service Provision Point* (SPP). The SPP may be replicated; the number of replications is given by a replication factor. Its counterpart, the interface that accesses services of an SPP is called *Service Access Point* (SAP). SAPs can be replicated as well but there rarely is a need to. The SPP and the SAP, each has a protocol associated with it that determines the interface type. Similar to a binding, a SPP and a SAP are connected to each other by a *layer connection*. By definition, the protocol of the SPP is the conjugated protocol of the SAP. Like end ports, SPPs and SAPs are directly available to the behavior component of an actor. All individual layer connections between two actors including their SPPs and SAPs are said to establish a *layer contract* between both actors.

A very simple example of a layering relationship is shown in figure 2.13. Note that most of the language concepts for layering have a textual representation only and do not show up in diagrams. There is just a layer connection symbol, an arrow pointing from the SAP side to the SPP side, which always has to be complemented by a textual description that specifies details of the involved SAPs, SPPs and protocols. Names attached to the layer connection symbol begin with a small letter

by convention.

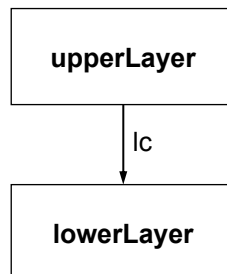


Figure 2.13: Layer connection, see also [SGW94, p.202]

So far, SAPs, SPPs and layer connections do not differ that much from ports and bindings except for the naming. That is what makes it so hard to identify a semantic difference. Though, since actor classes can be nested through actor references, some conventions are required how “inner” SPPs and SAPs of an actor reference can be used from the “outside” of the containing actor class.

An “inner” SPP can be made accessible by the actor class in form of a *relay SPP*. It is the same technique to selectively export interfaces from inside-out as we already know it from the relation between a reference port and a relay port. Visually, a connection symbol, now called *export connection*, is drawn that points from the border of the actor class towards the actor reference; it encompasses *all* SPPs that are *exported* to the actor class, see figure 2.14. For details, which individual SPPs of the actor reference are exported, the accompanying textual specification is to be consulted.

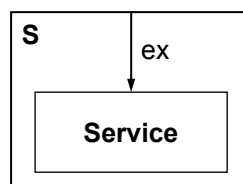


Figure 2.14: Export connection, see also [SGW94, p.205]

In contrast to “inner” SPPs, an “inner” SAP that is unfulfilled (meaning it is not matched against an SPP by a layer connection) is automatically exported to its “outer” actor class and becomes part of the interface of the actor class. Consequently, there is no graphical export notation for SAPs. Automatically exported SAPs remain totally hidden in ROOM diagrams.

The reason for this special treatment of SAPs is that SAPs in ROOM are perceived as the services of an actor it requires for its implementation. All SAPs are

regarded as vital for the functioning of the actor, so that selective export is no option. All unfulfilled SAPs of an actor class (which includes automatically exported SAPs of inner actor references) are said to specify the *virtual machine* of that class.

Note: The understanding of a SAP and a SPP in ROOM differs considerably from the conception of a SAP we derived from OSI in an earlier section. To avoid confusions we will explicitly talk about the ROOM SAP and the ROOM SPP when we refer to the language conception as introduced by ROOM. Otherwise, we mean the SAP concept in the OSI sense.

2.3.2 Behavioral Elements

ROOMcharts, Scheduler

We already mentioned the behavior component of an actor. In ROOM, behavior is specified in form of state machines, so-called *ROOMcharts*, a variant of HAREL'S statechart formalism [Har87]. Actors in ROOM are reactive objects with their own thread of execution, which is a typical characteristic for real-time systems. All incoming messages at the behavior component are *events* that may trigger a *transition* to leave a *state*, perform some *action* and enter the same or another state. For a state, entry and exit actions can be specified. Actions are specified in a detail level language such as C [KR88], C++ [Str00] or Java [AGH00]. A *guard* (a boolean condition) can be attached to a transition, that prevents the transition from firing if the condition evaluates to false. The concept of *composite states* enables the modeler to nest states within states. Once all actions have been executed (ROOM follows the "run-to-completion" processing model), the actor "falls asleep" waiting for further events to process. Since incoming events are queued, the actor may immediately come busy again until the event queue is empty. Events can also be deferred i.e. the processing is postponed. Message priorities change the order of event processing, usually to "the more important, the more up front in the event queue". In principle, the scheduling semantics of the *scheduler* can be adapted to any other scheme. ROOM is quite flexible in that respect to cover a wide range of real-time applications. For example, time-based scheduling ("the more urgent, the more up front in the queue") may be an alternative.

Since behavioral specifications are of minor interest for us, we do not deepen the introduction of ROOMcharts. In this work, our primary focus is on the architecture level of system design. Therefore, the structural elements of ROOM deserved greater detail of explanation. We just take it for granted that the reader is familiar with state automata and can read UML statechart diagrams [OMG01]. As a service to the reader, we will uplift ROOMcharts to UML statecharts; the notational differences are minor. In case that there are UML statechart features

that deviate significantly from ROOMcharts we will leave a note to the reader.

Data Classes

Complex data structures can be modeled using the concept of *data classes*. Data classes correspond to traditional classes: they define data and methods that operate on them. In contrast to actors, data objects do not have their own thread of control; they are extended state variables that are encapsulated within the actor and are accessible by the behavior component. Typically, data classes are based on classes provided by the detail-level programming language. That means within an actor the modeler can use and stick to a quite traditional object-oriented design paradigm.

In addition to their role as variables, data classes are used to define the data carried in messages. Remember that a message consists of a name, a priority and data, or more precisely, a data object. This single data object is an instance of a predefined or user defined data class. The basic requirement put on data objects is that they must be serializable for message transfer by the ROOM Virtual Machine.

2.3.3 Model Execution

In principle, there are two possibilities to execute ROOM models: (a) the model is accompanied by an interpreter called the *ROOM virtual machine*, which interprets the model; (b) the elements of the model are mapped to their functional equivalents in the target environment, which usually is a real-time operating system. Here, we skip the compilation of ROOM models to a target environment and briefly touch upon the ROOM virtual machine.

ROOM Virtual Machine

The ROOM Virtual Machine (VM) interprets ROOM model specifications; it is a hypothetical platform implemented either in hardware or software and executes ROOM models. The basic relation of a ROOM model specification (the application) and the ROOM VM is shown by a ROOM diagram in figure 2.15. The ROOM VM itself uses basic services (exception, memory and clock services) of the target environment and provides four services to the application (the top-level actor class): the *frame service* (*frame*) is responsible for maintaining the run-time structure of actor class incarnations; this includes actor incarnation and destruction, imports and substitutions; the *communication service* (*comm*) is (a) responsible to maintain, i.e. insert and delete connections between actor interface components (ports, SAPs and SPPs) and (b) takes care of delivering messages via established connections to destinations; the *timing service* (*time*) is basically an

interface to realize timeouts; the *exception service* (excpt) to realize exceptions (e.g. to raise an incarnation exception).

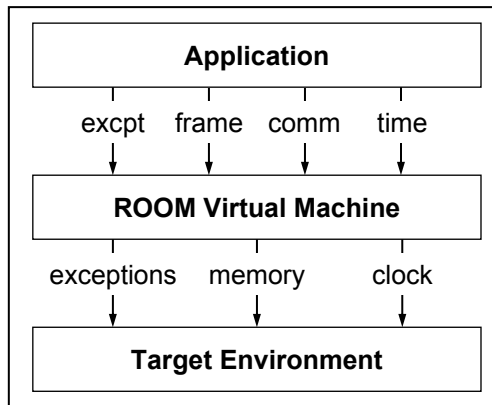


Figure 2.15: Relation of the application, the ROOM VM and the target environment, see also [SGW94, p.325]

All actor classes referenced by the application top-level actor class make use of these four services. Since the corresponding SAPs are given by default they are not made explicit in ROOM diagrams. So, a ROOM actor class has a set of peer interfaces (ports), it provides services at SPPs, and uses services of a virtual machine. The actor's VM is given by its unfulfilled SAPs plus the ROOM VM SAPs.

Note that the ROOM VM can be specified using ROOM. This circular specification technique is not an uncommon approach for the design of (modeling) languages (see e.g. [ASS96] and [OMG01]).

2.3.4 ROOM Tools

There are some few CASE (Computer-Aided Software Engineering) tools available that support the creation and execution of ROOM models. Two of them (Rational Rose RealTime and trice) are commercially available; another one, JRealTime, spurred around for some time in the World Wide Web (WWW) but is not available anymore. A fourth one, PyROOM, has been developed for research purposes; its successor PyROOM++ is used in this work.

Rational Rose RT

The authoritative ROOM tool, ObjecTime Developer [Obj98], is a full implementation of ROOM as described in the ROOM book; it is the result of over 120 man-years of research and development since 1986 into the use of executable,

object-oriented models for producing large-scale embedded systems [Bra96]. ObjecTime, a Canadian company, was bought by Rational Inc. (now owned by IBM) in the late nineties. The tool ObjecTime Developer has been integrated in Rational's product line and is now sold under the name Rational Rose RealTime.¹³ It has been slightly adapted to the UML look-and-feel and extended by some few UML features like use cases. There are some activities ongoing, mainly driven by Rational, to incorporate the ROOM language in the forthcoming UML 2.0. Rational Rose RT supports common languages like C, C++, Ada, and Java and a variety of real-time operating systems. The use of the tool requires extensive training and practice until it can be used productively.

Trice

Trice is a remake of ObjecTime Developer developed by protos GmbH, a german company located in Munich.¹⁴ Trice supports a subset of ROOM (there are currently no SAPs/SPPs supported) and targets for a market niche in the embedded systems area. Thus, there is some focus on optimized run-time behavior for specialized real-time operating systems. Due to its limited feature set, trice is intuitively to use and quick to learn.

JRealTime

JRealTime by Dolbec Consulting is another re-implementation of ROOM. It supports only very basic structure language conceptions of ROOM (actors, ports, and bindings), is written in Java and accepts Java as an action language only. The tool binary was available for free download in the year 2000; the tool and its home page has disappeared from the web ever since. Since the tool also has had some severe bugs it cannot be considered as a serious ROOM implementation; it has been listed for completeness only.

PyROOM

PyROOM is the author's implementation of ROOM in Python [Lut01]. PyROOM has been developed as a research prototype to implement and test ROOM language extensions. Neither Rational Rose RealTime nor Trice are extensible in that respect, which was the main show stopper to use them in the E-CARES research project. PyROOM has no graphical user interface, the actor behavior has to be completely specified as a Python program, and it supports ROOM structural elements just to the extend needed for an early prototype. In addition to that

¹³For more information see <http://www.rational.com>.

¹⁴See <http://www.protos.de>.

PyROOM served as a proof of concept that a dynamically typed and interpreted language enables rapid model prototyping in ROOM, which suits much better the needs of early design phases as tools such Rational Rose RealTime do. More details about PyROOM++, the successor of PyROOM developed for this work, can be found in chapter 6. PyROOM++ fully supports the ROOM language including the extensions proposed in this work.

2.4 Mathematical Formalism

To define some key principles in this work we require the precision of a mathematical formalism: an algebra. In this section, we give a brief summary on the formal approach of components and streams as introduced by BROY in [Bro98a, Bro96, Bro93, Bro98b] before we move on to apply the mathematical conception on the notion of distribution and layering in subsequent chapters. The notational and algebraic conventions are based on FOCUS [BS01]. While ROOM and FOCUS were not specifically designed for each other, the mapping of simple ROOM models to FOCUS and vice versa goes without problems.

2.4.1 The Concepts of Streams, Channels, and Components

Informally, a component is typically defined as a “physical encapsulation of related services according to a published specification” [Bro98a, p.131] (but see also [BDH⁺98]). Note that we would like to generalize the quote and speak of a *logical* encapsulation of services; the demand of a physical materialization is an unnecessary constraint. The underlying concept is the idea of a component which encapsulates a local state or a distributed architecture. Based on this idea, the mathematical notion of a component is rather straight: it relates streams of messages (or actions) of input channels to output channels. All required concepts are defined subsequently.

Untimed Stream An *untimed stream* over the set M is a finite or infinite sequence of elements from $M = \{m_1, \dots, m_i\}$ with $i \in \mathbb{N}$. The elements of M represent messages or alternatively actions. For the sake of brevity, we will refer to messages in the following. To be concrete, let us assume that M consists of two messages $M = \{m_1, m_2\}$. We now construct an infinite number of streams out of M . For example, the sequences $\langle m_1 \rangle$, $\langle m_2, m_1, m_1 \rangle$, and an infinite series of m_2 messages $\langle \dots, m_2, \dots \rangle$ represent each valid streams over M . The *empty stream* is denoted by $\langle \rangle$.

The set of all finite streams which can be formed over the set M is denoted by M^* . M^∞ , on the other hand, denotes the set of all infinite streams over M . Putting the sets M^* and M^∞ together results in the set of all possible streams over M :

$$M^\omega =_{\text{def}} M^* \cup M^\infty$$

Timed Stream To introduce the notion of time, we suppose a discrete time model with time intervals of equal length. By adding so-called *time ticks* (represented by a \surd symbol) in a stream, we indicate the progress of time and have introduced the concept of a *timed stream*. For example, the transmission of m_1 in the first time interval prior to the first tick, and the transmission of m_2, m_1 , and m_2 (in exactly

that order) in the next time interval between the first and the second tick, is noted as the timed stream $\langle m_1, \surd, m_2, m_1, m_2, \surd \rangle$.

Accordingly to untimed streams, the set of all finite timed streams over M is denoted by M^* . That means, there is a finite number of ticks in each individual stream. M^∞ denotes the set of all infinite timed streams over M with an infinite number of ticks. The conjunction defines the whole set of all finite and infinite timed streams over M :

$$M^\omega =_{\text{def}} M^* \cup M^\infty$$

Note that a stream represents a history of communication, while M^* , M^∞ etc. represent a set of communication histories. Whenever we would like to express that a stream s represents a concrete instance of e.g. a timed infinite communication history, we write $s \in M^\infty$. As we will see, the notion of communication histories allows us to describe the specification of a component in a very compact manner.

Channel A (directed) *channel* is an abstract concept of a media that transfers messages from a sender to a receiver; the transfer is uni-directional, immediate, faultless and the order of messages is preserved. A channel consists of a channel name (also called *identifier*) i and an associated channel type T . The relation identifier/type is usually denoted as $i : T$. In general, a type T is given by a set of messages, which can be potentially sent over the channel. For example, the set $\mathbb{Bit} =_{\text{def}} \{0, 1\}$ may stand for type T of the channel identifier i . Thus, $\text{type}(i) = \mathbb{Bit}$. Formally, the type assignment is given by the mapping

$$\text{type} : i \rightarrow T$$

The timed stream s , $s \in \mathbb{Bit}^\infty$, is a valid history of the space of all infinite “bit-ed” communication histories transferable over the channel i .

Channel Valuation Specifically, the messages of a stream s associated to a channel identifier i have to be elements of the channel type $\text{type}(i)$, $i : \text{type}(i)$, meaning the *channel valuation* has to be fulfilled:

$$\forall j \in \{1, \dots, \#\bar{s}\} : \bar{s}.j \in \text{type}(i)$$

Here, \bar{s} stands for a stream with all ticks removed from s , $\#s$ is the length of s , and $s.j$ returns the j th element of s . The channel valuation can be guaranteed for an e.g. infinite, timed stream s by

$$s \in \text{type}(i)^\infty$$

Component Seen from a black-box perspective, a *component* can be specified by (a) its syntactic interface and (b) its behavior which can be observed at the interface.

The syntactic interface is given by a list of channel names, which we refer to as input/output identifiers (I/O identifiers), and a list of associated channel types. For any component specification S , by $i_S = (i_1, \dots, i_n)$ and $o_S = (o_1, \dots, o_m)$ ($n, m \in \mathbb{N}$) we denote its list of input and output identifiers, respectively. $I_S = (I_1, \dots, I_n)$ and $O_S = (O_1, \dots, O_m)$ are the corresponding lists of their types. For the association channel identifier to channel type, we also write $i_1 : I_1; \dots; i_n : I_n$ (or $i_S : I_S$ for short) and $o_1 : O_1; \dots; o_m : O_m$ (or alternatively $o_S : O_S$). We refer to $(i_S : I_S \triangleright o_S : O_S)$ as the *syntactic interface* of S ; $(I \triangleright O)$ is a shortcut notation for that. To state that S is a specification with the syntactic interface $(I \triangleright O)$ we write

$$S \in (I \triangleright O)$$

and depict it graphically as shown in figure 2.16.



Figure 2.16: Specification S with the syntactic interface $(I \triangleright O)$

For the behavioral description of S we first need to associate type consistent streams to each of the channel names. In favor of a compact notation, the I/O identifiers also designate their respective streams, which represent communication histories; i.e. $i_S \in I_S^\infty$ (abbreviated for $i_1 \in I_1^\infty; \dots; i_n \in I_n^\infty$) and $o_S \in O_S^\infty$ (for $o_1 \in O_1^\infty; \dots; o_m \in O_m^\infty$) in case of infinite timed streams. The black-box behavior of a component can now be described by predicates characterizing a subset out of all possible I/O histories

$$\mathcal{R}_S \subseteq I_S^\infty \times O_S^\infty$$

With B_S representing the body of the specification S , this set of I/O histories forms a relation \mathcal{R}_S called the *I/O behavior* of S and is defined by

$$(i_S, o_S) \in \mathcal{R}_S \Leftrightarrow B_S$$

Note that the definition of the I/O behavior \mathcal{R}_S is general enough to include non-deterministic system behavior as well. An input history may have more than a unique response of output histories. For causality restrictions, [BS01] can be consulted.

Definition 2.1 (Denotation) For any timed elementary specification S written in the relational style, we define its denotation, written $\llbracket S \rrbracket$, to be the formula

$$i_S \in I_S^\infty \wedge o_S \in O_S^\infty \wedge B_S$$

□

2.4.2 Composition with Mutual Feedback and Behavioral Refinement

Some more definitions come in handy to work with arrangements of components. If some output channels of one component are input channels of another component and vice versa, we speak of *mutual feedback*.

Definition 2.2 (Composition with Mutual Feedback) *Given two specifications $S_1 \in (I_1 \triangleright O_1)$ and $S_2 \in (I_2 \triangleright O_2)$ with disjoint sets of output identifiers $O_1 \cap O_2 = \{\}$, $l = (I_1 \cap O_2) \cup (I_2 \cap O_1)$ stands for a subset of channel identifiers of O_1 and O_2 , which are input to I_2 and I_1 , respectively. The same type L is assumed for the affected internal channel identifiers in l . Composition with mutual feedback $S_1 \otimes S_2$ is then defined as*

$$\llbracket S_1 \otimes S_2 \rrbracket =_{\text{def}} \exists l \in L^\infty : \llbracket S_1 \rrbracket \wedge \llbracket S_2 \rrbracket$$

□

Behavioral refinement allows us to add step by step properties to a specification, while it is guaranteed that any I/O history of the refined specification (the more concrete specification) is also an I/O history of the given specification (the more abstract one).

Definition 2.3 (Behavioral Refinement) *Let S_1 and S_2 be specifications with the same syntactic interface. The relation \rightsquigarrow of behavioral refinement is defined by the equivalence*

$$(S_1 \rightsquigarrow S_2) \Leftrightarrow (\llbracket S_2 \rrbracket \Rightarrow \llbracket S_1 \rrbracket)$$

□

Composition with mutual feedback is modular with respect to behavioral refinement.

Composition with feedback is powerful enough to express all kinds of communication connections between components provided the channels of the components are consistently named. In many applications we want to connect only a subset of the input and output channels of one component with those of another one. To do that we use the notion of a *connector*.

Definition 2.4 (Connector) *Let $S_1 \in (I_1 \triangleright O_1)$ and $S_2 \in (I_2 \triangleright O_2)$ be given components. A connector C for S_1 and S_2 is a component in $(I'_1 \cup I'_2 \triangleright O'_1 \cup O'_2)$ where $I'_1 \subseteq I_1$, $I'_2 \subseteq I_2$, $O'_1 \subseteq O_1$, $O'_2 \subseteq O_2$. Its composition is defined by*

$$\exists I'_1, I'_2, O'_1, O'_2 : \llbracket S_1 \rrbracket \wedge \llbracket C \rrbracket \wedge \llbracket S_2 \rrbracket$$

and we then write

$$S_1 \leftarrow C \rightarrow S_2$$

□

For simplicity we assume here and in the following that all channels are named in a way such that there are no name conflicts.

2.5 Summary

When we looked upon the reference models of OSI and TCP/IP, we saw that stacking protocols of layers of communication is the combining mechanism that underlies OSI and TCP/IP. While the number of protocols may vary and their role in the stack may differ, they share the basic design principle. We noted an biased view determined by an implementation perspective that emphasizes vertical communication over horizontal communication. We derived an unbiased, harmonized set of concepts for both models:

- Horizontal or intra-layer communication goes via *connection endpoints*; the communication relation may be *connectionless* or *connection-oriented*.
- The communication relation is specified in form of a (*communication*) *protocol*. Sometimes we will refer to communication protocol messages as Protocol Data Units (PDUs).
- Vertical or inter-layer communication goes via (*service*) *interfaces*, also called *service access points*.
- The service relation is specified in form of a *service protocol*, often also by means of *service primitives*. We will also refer to individual service protocol messages as Service Data Units (SDUs).

Together with an investigation on the general structure of real-time systems we concluded that our modeling approach spans along three principles of design:

- distribution
- layering
- types of communication

First is the notion of *distribution*: If two entities are remote to each other but require collaboration, they need to bridge the separation by communication. A precise definition and a thorough discussion of its implications are subject of chapter 4. Next is *layering*: Layering concerns the realization of a remote communication by the services offered by another set of distributed entities. We will provide a definition of layering and a detailed discussion of it in chapter 5. Finally, the *type of communication* is related to who has a Controlled Domain Model (CDM) of whom. We will deepen this issue in chapter 3.

The treatment of real-time systems also motivated basic language concepts, which we manifested with the introduction of ROOM. ROOM is based on the notion of an actor that can be composed of other actors and communication bindings.

Besides direct peer communication, ROOM also supports layered communication. The investigations in the next chapters will unveil to which extend ROOM's language concepts are appropriate for modeling communication systems or not. Possible language extensions are discussed in the following chapters and are summarized in chapter 6.

For the purpose of formal definitions we introduced an algebra, whose building blocks are streams, channels and components; they are close to the concepts in ROOM.

Now, we are equipped with a set of tools that help us master the challenge of the next chapters.

Chapter 3

Types of Communication

This chapter's postulate is that the aspect of control is a key principle underlying the design of telecommunication architectures. Control helps us determine types of communication relations and has an important consequence on modeling such relations.

In section 3.1 we explain why the commonly used distinction in client/server and peer-to-peer communication models is not satisfying. The aspect of control is a better way of distinguishing roles and communication types. So, what is control? Which side in a communication relation controls whom? What are the implications of that? Answers on these questions are given in section 3.2; there, we will identify three types of communication and propose a notational extension to ROOM. Based on the insights gained, consequences of this classification are studied on services provided in a telecommunication system and exemplified on some "real" protocols: section 3.3 covers communication services and section 3.4 covers resource control services. As a result, a first set of elementary patterns for our architecture modeling approach is summarized in section 3.5.

3.1 Problems with the Client-Server and the Peer-to-Peer Communication Model

Opening a textbook about distributed computing, one is soon confronted with the client-server model and client-server architectures. The client-server model is a natural consequence of the revolution that moved us away from centralized systems (or single processor systems) to distributed computers connected by a network. A *client* is a process that requests a service from a *server* (another process implementing the service, e.g. a database service) by sending it a request and subsequently waiting for the server's reply (see [TvS02, p.42f.] and [Tay98]). This interaction is also known as *request-reply behavior*. Client-server architectures suggest a physical organization of client-server applications, they are called *multi-tiered architectures* [TvS02]. If there are only two kinds of machines, one speaks of a *two-tiered architecture*. For example, one machine hosts the user interface, the other the application including the database. In a *three-tiered architecture*, the three application parts are distributed over three hosts. The user interface acts as a client on one machine, the database as a server on another machine, and the application resides in the middle on a third machine, acting as a server towards the user interface and as a client towards the database.

The problem with the client-server model is that it usually only addresses the application level. However, complex communication systems as we discuss them are organized on many levels, the application level is just one of them. Does it make sense to use the terminology "client/server" on other levels as well? We doubt it – at least, it is not that simple anymore. We already saw in chapter 2 that there is an alternative to the request-reply behavior: it is the *indicate-response* behavior. In [Mut01] we find even some more categories of behavior. None of them matches to the simple client-server model anymore.

Besides that, even when we accept the request-reply behavior for a moment, we may feel uncomfortable with the client-server model. There is a difference: a client requesting the time from a server and a client requesting an alarm service. In the former case we may just retrieve a string containing date and time; in the latter case, we get access to some sort of resource we continue using to set and get alarms. The client-server model only speaks of a *service* provided by the server and does not make a difference between simple operation requests and resource requests.

During the last years, another communication model gained attention in networked systems, so called *peer-to-peer distribution* [TvS02, p.53]. Such networks try to eliminate the need of servers as a shared source of services [Ora01]. One reason for such a model is to avoid hot spots of network load. Servers are natural sinks of service requests, which focus the network load on few locations and

3.1 Problems with the Client-Server and the Peer-to-Peer Communication Model 73

lines. Peer-to-peer networks tend to balance the traffic much better at the cost of having a higher traffic overall. Another reason, often regarded as more important, is a greater degree of anonymity and privacy. Without servers, there are no single organizations anymore that run and maintain the server(s) and, in principle, have access to all the data stored at the server side. On the other hand, in peer-to-peer networks pieces of information may easily get lost, simply because the peers holding information are not connected to the network.

The peer-to-peer model is not lesser problematic as a communication model than the client-server model is. It also targets the application level. What about the lower levels? Is it meaningful to talk about peer-to-peer communication on lower levels? Another issue is: Even though there are no distinct permanent servers anymore that does not mean that two peers cannot temporarily work in a client-server mode. Most likely we will identify several modes of communication among peers. In some cases, the communication will clearly follow the request-reply or indicate-response behavior; in other cases, we may observe who has initiated the communication but will not identify any specific behavioral pattern besides the fact that both parties stick to the rules of a communication protocol.

That is a confusing result. Does that mean that the distinction in client-server and peer-to-peer is worthless? No, it does not. The problem is that an organizational pattern and roles of communication are subsumed under the same term – and literature is not clear on that. If centralization of services shapes the structure of a network, it is reasonable to name the centralized unit *server* and the accessing (remote) entities *clients*. However, the request-reply behavior is an idealization of that organizational pattern and is often violated in reality. On the other hand, if decentralization of services shapes a network structure, it is useful to call the communicating entities *peers*. Temporarily, the peers may establish any form of a communication relationship.

This chapter's mission is to investigate different types of communication relationships and to introduce other names than client-server and peer-to-peer for them; we will reserve these terms for the above mentioned organizational styles. We will gain clarity from that and use the categorization of communication types in later chapters. Note that the type of communication is independent of the aspect of distribution. We will add the dimension of distribution later on in another main chapter.

3.2 Alignments of Control

The distinction we are going to make is almost a classic: *control flow* versus *data flow* [DeM78]. We will tackle the problem of control from the ground up (What is control? Who controls whom?) and make an interesting observation: there is more than a *control-oriented* and *data-oriented* type of communication. A third type, *protocol-oriented* communication, is not just a mixture of both, but a type in its own right. We derive these three types from going through all possible variants of control: only one side in a communication relationship exerts control, both sides exert control, or none of them. After that investigation, we turn back to the telecommunication domain in section 3.3 and section 3.4 and study types of communication to be found on various interfaces in a telecommunication system.

3.2.1 What is Control?

A simple scenario of a control-oriented communication relationship is depicted in figure 3.1 in form of a ROOM diagram. It consists of a controlling actor, the *controller*, on the left hand side and a controlled actor, the *controllee*, on the right hand side. The controller and the controllee are connected by two bindings for communication. The ROOM protocols are defined from the perspective of the controllee, which is indicated by the “black” ports on the controllee’s side. The set of valid messages is annotated close to the bindings.

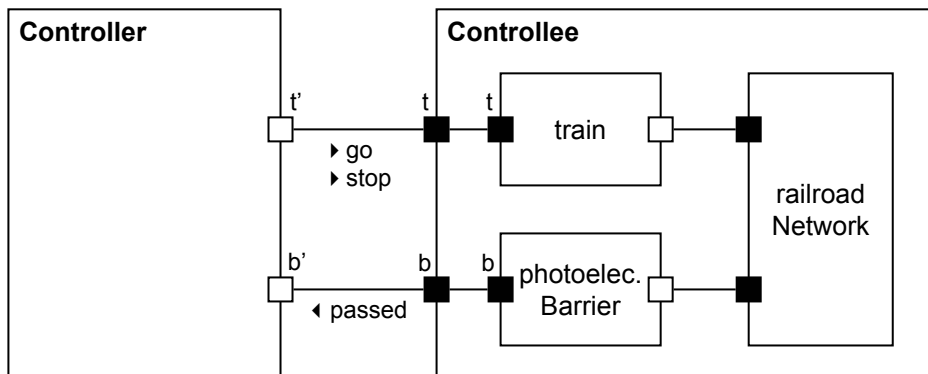


Figure 3.1: A simple controller/controllee scenario

In our simple scenario, the controllee refines to a minimalistic model of a railroad with the train being the resource under control. In addition to the train, there is a photoelectric barrier attached to the railroad network that sends out a passed message as soon as a train passes by. We do not further elaborate details of the train/network/barrier relation; we just assume that the network is a closed,

circled stretch of line, so that the train must pass the photoelectric barrier each round.

Via the controllee's port *t* the train can be set into motion with the message *go* and halted with the message *stop*. Since we do not want to work with real world entities, we use a simple state model of the train, see figure 3.2. The initial state is *OFF*. In that state, event *go* lets the train move (*move*) and changes the state to *ON*. The event *stop* causes no action, the train remains in state *OFF*. If in state *ON*, the train halts (*halt*) and changes to state *OFF* if the event *stop* occurs. Event *go* in state *ON* is without any effect. In this example, we assume that messages and events relate to each other in a one-to-one manner as is usually the case in *ROOM*. Message *go* triggers event *go* and message *stop* triggers event *stop*.

This is the simplest possible model of a resource. The state space consists of only two states, which can be represented by a single bit.

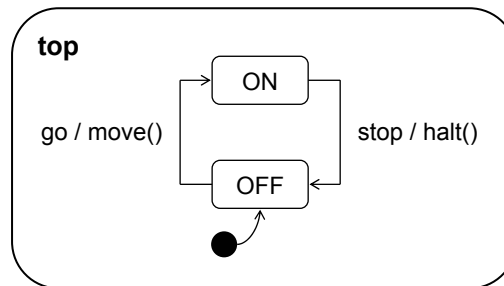


Figure 3.2: State model of the train resource

In control theory, *control* is defined as follows: “To control an object means to influence its behavior so as to achieve a desired goal” [Son98, Chap.1]. A direct implication of this definition is that the controller needs to have an understanding of the functioning of the object – at least to the extent that concerns the observable behavior of the object. Observable behavior refers to (a) feedback and notification messages sent from the object to its controller, and (b) messages sent from other objects to the controller that let the controller deduce information about the controlled object. These other objects have to be indirectly or directly related to the object under control.

Applied on our example: If we as the controlling actor would like to steer the train, we need to have some sort of representation of the train's state space and assume its current state. In chapter 2 (page 45), we reserved the name *Controlled Domain Model (CDM)* for such kind of representation. We have to assume the current state since we cannot know it for sure. For example, the engine might become overheated and is in danger to break of which we get no direct notice since we are not equipped with a heat sensor in our model. Without an internal representation of the train and its assumed state, and without a model of related

objects, the controller cannot meaningful control the train. We have to memorize that the train currently is, for instance, in state ON (since go was our last message we sent), so that we can stop the train at a certain point in time. For example, because we got an indication by another resource, say the photoelectric barrier, that the train passed a defined location. If we are not aware about the probable state of the resource and do not know how we could affect a state change, we loose the capability of directed impact on the behavior of the resource. Then, we are not in control of the resource.

Control is the *directed* exertion of influence on the behavior of something or someone. We call the controlling party the *controller*, the controlled party the *controllee* and the type of communication relationship *control-oriented*. Note that the definition implies that the controller has some sort of a model, a *Controlled Domain Model (CDM)*, of the controllee.

But how can we be sure that it is the controller that controls the controllee and not vice versa? Are there some deeper semantics of control? In the next subsection we will cover both questions.

3.2.2 Who controls whom?

The definition of control is clear, but still: If an external observer monitors the messages exchanged between two communicating parties, can the observer draw any conclusions on who controls whom? Are there any criteria on which we can uniquely decide if there is a control-oriented communication relationship or not? Who is the controller, who the controllee?

Indisputable is the prerequisite that the capability to send out messages must be given. Otherwise the basic precondition to exert influence is not given. Though, the conditions are not as simple as it might look like. In the example it is only the controller, who sends messages to the train; nonetheless, this is no unique criterion for control. If we would observe the controller sending out go/stop messages randomly, there would be an impact on the train for sure, but we would not speak of directed, meaningful control (which is demanded in the definition): The controller does not seem to know (or does not want to know), what he impacts and how he impacts it. Even on the appearance of a single, unnecessary go message in state ON we would become suspicious and doubt on the capabilities of control of the controller.

We have to demand a correlation between messages from the controller and the behavior of the controllee. A message that does neither influence current nor future behavior of the resource does not change anything; the message is without influence, it is not correlated and herewith superfluous. This contradicts the understanding of directed control.

Let us assume correlation between the controller's messages and the train's behavior and drop the photoelectric barrier for a moment. We will observe the controller alternately sending out go/stop messages to the train. Although we do not know the criteria on which the controller lets the train stop and go (without the photoelectric barrier and no other feedback, the controller is "blind"), we as an external observer see that it makes perfect sense. A go sets the train into motion, a stop brings it to a halt. In this special case the control-oriented relationship is unique and without question; unidirectional messages and message/behavior correlation leave no other option.

The appraisal of the control relationship becomes more difficult, if we take the feedback of the photoelectric barrier into consideration. Assumed that the controller halts the train as soon as the train passes the photoelectric barrier. The controller lets the train take a rest for some duration t_1 , which appears to be randomly chosen, and then activates the train again. Now, who controls whom? Intuitively, it is the controller who controls the train, but is that correct? Is it not so that the train after some distance stretched, no matter whether driven fast or slow, induces the controller to stop it? For an external observer, message stop appears right after passed. If stop is an immediated reaction on passed, is it the train controlling the controller?

As a matter of fact, from an external viewpoint we cannot draw a clear conclusion anymore. The relation of cause and effect is not always resolvable. Message passed is instantly acknowledged by stop, followed by a go after some time t_1 , see the sequence chart on the left hand side in figure 3.3. If one exchanges the names of the messages with a new semantic meaning, i.e. if one changes passed, stop and go to randomTime, begin and end, the message sequence tells a completely different story (see right hand side of figure 3.3). In this new version, the controllee seems to use a random-time generator. It looks more like that the controllee uses a service of the controller, there is no control relationship anymore in the sense it has been before. The controllee does not need a CDM of the controller.

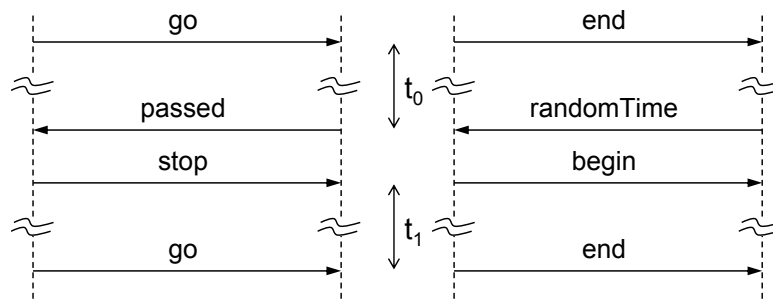


Figure 3.3: Sequence diagrams of the message exchange between controller and controllee

The lesson to learn is that, in general, control-oriented communication cannot be deduced by an external observer. In other works, black-box specifications hide the aspect of control. Without any knowledge about the internal of an actor it is often impossible to say whether the actor can or does exert directed control. For control, the actor needs to have an internal representation, a Controlled Domain Model (CDM), of the environment that is specified by the corresponding ports. The controllee, on the other hand, does not need to have any representation of the controller. The knowledge, whether an actor maintains a CDM or not, requires a grey box specification style. We need to know some details of the actor's implementation in order to exhibit the fact of control.

Equipped with this basic insights about the nature of control, we now go through all possible variants of control-orientation in a communication relationship. Control may be exerted from one party only, from both parties, or none. We presume static communication relations in the sense that the side of control does not change over time.

3.2.3 One-Sided Control-Oriented Communication

In *one-sided control-oriented* communication, only one party of a communication relationship maintains a state model of the opposite side in order to exert control. The previously discussed railroad model is an example of that type of communication. The situation is diagrammatically depicted in figure 3.4: the actor on the left hand side, the controller, has a CDM of the actor on the right hand side, the controllee. Of course, *controller* and *controllee* are role names with regard to a specific binding.

The controller's model of the controllee must by no means be a model of the internals of the controllee. It is (a) a question to which extent the controller has precise or assumed knowledge about the controllee's behavior and state, and (b) a matter of viewpoint: the controller often sees via its ports just a fraction of the whole that makes up the controllee. We call an actor's model of its implementation a *Domain Model (DM)* as is shown in figure 3.4, of which the CDM is a viewpoint of. We can also see from figure 3.4 that the domain model may include other CDMs based on other communication relationships. Hence, CDMs are viewpoints on other DMs. Viewpoint modeling is a result of control-oriented communication – and is a very powerful abstraction technique. A hierarchy of control-oriented communication relations can be used to combine CDMs, to abstract them to new DMs and to provide higher-level services.

From a specification point of view it is sufficient to specify the CDMs only. An actor's DM, a model of its implementation so to speak, is determined by the set of its own CDMs and the CDMs addressed as expectations on it via control-oriented communication relations. From a practical standpoint, selected DMs might be



Figure 3.4: One-sided control-oriented communication

specified prior to the CDMs. Sometimes – and that is experience from practice –, it is easier to come up with a CDM when one has gained clarity about the DM the CDM is a viewpoint of.

In any case, the DM is a model of the whole, the CDM a model of a fraction of the actor’s implementation on the next level of granularity. An actor can be refined and modeled in two different ways: *component-oriented* or *object-oriented*. It is a matter whether one would like to stick to a message-oriented communication paradigm or switch to a method-oriented communication paradigm. In ROOM, both styles are supported and can be used in combination. However, while component-orientation refers to refinement in form of actor references (which is ROOM’s basic decomposition feature), object-oriented modeling with ROOM’s data classes is a neglected area. It is even barely scratched in the ROOM book.

The discussion about component-oriented and object-oriented refinement is a topic of general interest and not specifically tied to control-orientation. However, we use the railroad scenario to exemplify the two approaches and stress some implications of control-orientation.

Component-Oriented Modeling

According to our argumentation, the controller needs to have a representation of the state space of the controllee. The simplest solution to that is to design a finite state machine for the controller’s behavior component that digests all incoming messages, remembers the assumed state of the controllee and acts according to some plan describing the control behavior. Figure 3.5 shows one possible state model. For simplicity, we do not mind about the strategy that internally triggers off the next message. Of course, the state model reminds very much of the state diagram we specified for the train resource (figure 3.2). But, and that is a major drawback of such a state model, the control behavior and the state space model of the controllee are interwoven in a single state diagram. If possible, we should separate concerns.

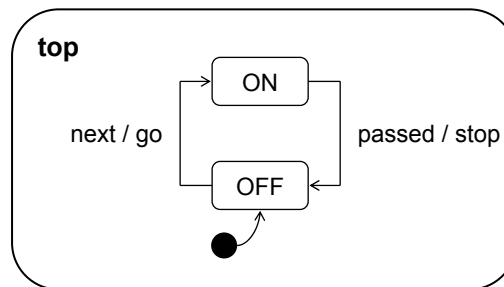


Figure 3.5: State model of the controller

Alternatively, we could refine the actor's implementation and, as an option, outsource parts of the state space to an explicit model of the controlled resource. Structural refinement of an actor's implementation is done via actor references in ROOM. We call this style of modeling the internals of an actor *component-oriented*; it is ROOM's basic feature of decomposition. In figure 3.1 we already used that way of decomposing the controllee in a train, a photoelectric barrier and a railroad network. The actor references including their interconnections constitute the Domain Model (DM) of the actor.

A proposal for the controller's component-oriented domain model is shown in figure 3.6. It consists of a switch and a driver using the switch and the information received from port b'.

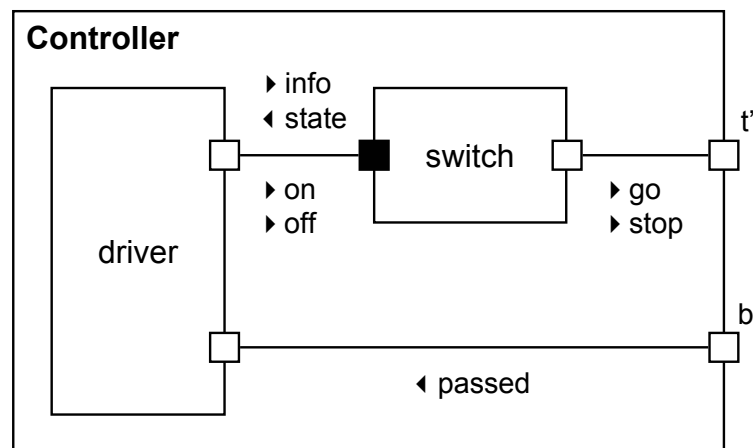


Figure 3.6: Domain model of the controller, component-oriented

The controller does not have a model of a train, of a photoelectric barrier or of a railroad network. The controller works just with the limited perception of the world as it can be perceived via the ports t' and b'. From the perspective of the controllee the switch is just an aspect of the train, an aspect published via port t.

That is what we mean by viewpoint modeling: The controller's CDM, the switch, is only a view of the controllee's DM.

The switch expects on/off messages and converts them to stop and go messages. We assume that the switch maintains an own state model based on monitoring the messages it converts. In addition, we added two messages, *info* and *state*, that enable the driver to query the current state of the switch. This technique is called *reflection* [DM95] and offloads the driver to maintain an own state model of the resource it controls. With reflection, the driver can also switch to a rule-based formulation of the control behavior. A rule could be "if message passed has been received and if switch is in state ON, then send message off". We could also add further features to the switch, such as a blocking mode that prevents the switch to react on on and off messages. This way, the viewpoint model switch of the resource train can be enhanced by features the controlled resource is not capable to handle.

Instead of reflection, another possibility is to define a synchronization protocol between the driver and the switch. This would also support the separation of control behavior and controlled state representation. More about synchronization as a technique for state space separation can be found in chapter 6.

Without reflective capabilities of the switch and without synchronization means with the driver, the driver would have to work with a state model similar to figure 3.5. In that case, the switch is more of an explication of the conceptual entities the controller assumes to be on the controllee's side rather than a representation of the controllee's state space.

All this modeling options for the CDM should make clear one point: there are many ways to model the CDM; integrated with or separated from the control behavior, with reflective or synchronized capabilities or without. From an engineering standpoint it is beneficial to have an explicit, discrete model of the CDM, so that the viewpoint of the entities under control becomes clear. If the CDM can be used in a reflective or synchronized manner, one can even separate control behavior and controlled state space tracking. Admittedly, the borderline that separates control behavior and the controlled state space is often fuzzy and is partially an arbitrarily decision of the modeler. Still, the advantage is that such a CDM can be used as a requirement on the domain model of the controlled actor and that we can aim for reuse. If designed with reuse in mind, a copy of the controller's CDM can be integrated as a component in the DM as well.

Thus, for components that exert control, we can (but must not!) structure our models in such a way that the distinction in a CDM and remaining parts remains visible. In figure 3.7 we have diagrammatically depicted such a structural pattern. The CDM's are represented by an actor reference each.

From a language design standpoint, the CDM is a consequent enhancement of the conceptual intent that underlies the notion of a port, or more generally, of

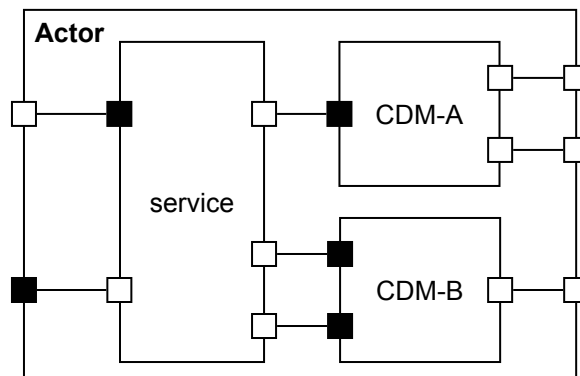


Figure 3.7: Structural pattern for actor that exerts control

an interface. In ROOM, an interface is not only a specification of how an external user can interact with the actor but also a specification of the assumptions the actor makes on its environment and, herewith, how it uses the environment. Yet, the specification level refers to messages and message schemata only. The CDM goes beyond the message level and adds the dimension of model views. One could think of moving the CDM on the border of the actor class specification as is indicated in figure 3.8 and regard it as a new interface type. Now, the CDM specifies the actor's view on a world it can control. This is a grey-box specification style of an actor that exhibits parts of its internal implementation to the outside.

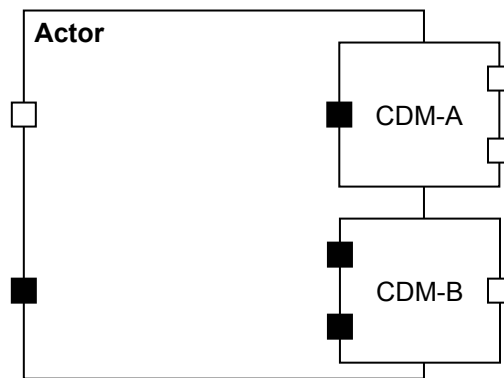


Figure 3.8: Grey-box specification of an actor

This is one way how one could extend the ROOM language. Since we do not want to enforce a specific way to design the CDM, we will go for a more conservative language extension and introduce a notation for control-orientation in subsection 3.2.6. Nonetheless, we will use this informal style of presenting an actor in a grey-box style from time to time.

Object-Oriented Modeling

If we use data objects as extended state variables, we can also approach the implementation of an actor from a completely different angle. Data objects are usually recruited from the language used for detail level action code such as Java or C++. Data objects can be accessed via the behavior component of the actor. The behavior component mediates between the message-oriented communication level of ROOM and method-calls of data objects. Message generation, on the other hand, is offered as a service by the ROOM virtual machine.

Instead to refine the actor in a component-oriented way, the alternative is to specify the domain model in form of a data class diagram. Data classes and data objects in ROOM correspond to classes and objects as known from object-orientation, see e.g. [Mey97].¹ Unfortunately, ROOM has not integrated a visual form of representing class diagrams (like known e.g. in the UML) in the graphical notation of an actor class. Figure 3.9 suffers from that fact; it uses an UML stylistic notation as a compromise.

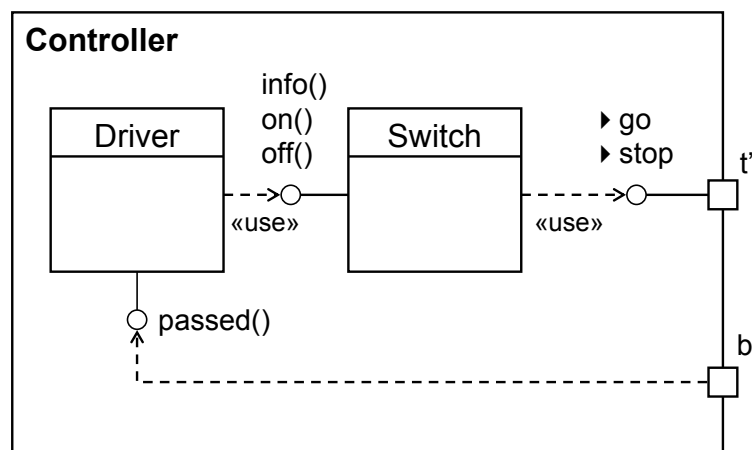


Figure 3.9: Domain model of the controller, object-oriented

Figure 3.9 is a conversion of figure 3.6 to an object-oriented modeling paradigm, which is based on method-oriented communication. Objects of Driver and Switch have to be instantiated by the behavior component e.g. at initialization time. Incoming messages like passed are consumed by the behavior component, the receiving object is determined and the corresponding method of the receiver is called. The function of the behavior component is to decode messages and dispatch method calls. Here, message passed results in a call of method passed() of

¹If data objects are not used as data to be conveyed in a message (that is where the name “data object” stems from), they can have methods that operate on the data. Outside the use of messages, data classes and objects are much better called just “classes” and “objects”.

a Driver object. Once this conversion step has been achieved, the objects can continue to use a method-oriented style of communication. In the example, a Driver object can invoke methods `on()`, `off()` and `info()` (with state as a return value) of a Switch object. If a Switch objects wants to send out a message, it uses the communication service of the ROOM VM to eject a `go/stop` message at port `t'`.

Remember that objects in ROOM are passive objects, i.e. the functionality of these objects is activated only in the context of their actor's execution thread [SGW94, p.150].

In general, the same design options apply as we discussed them for component-oriented refinement of the actor's domain model. We can target for a clear separation of the control behavior and the tracking of the controlled state space; we can use a reflective or a synchronization interface for that.

The advantage of object-oriented modeling is that OO models can be smoothly translated into an (object-oriented) detail level action language. Actor domain models and their behavior can be specified e.g. via UML diagrams, which are much more expressive (think e.g. of associations, multiplicity, stereotypes, interfaces etc.) than raw ROOM diagrams ever can be. A drawback is that the conversion of messages to methods is not optimally supported by ROOM:

- The behavior component of the actor needs to be prepared for object initialization, message decoding and method dispatching; there is no suitable infrastructure provided as a service by ROOM.
- If object `a` of actor `A` wants to communicate to object `b` of actor `B` (`B` is a direct neighbor of `A`) it cannot call a method directly from `b`. Object `a` has to send a message from `A` to `B` instead. This is time consuming and does not take full advantage of direct method-oriented communication but it preserves the exclusive thread control of actors.

As is often the case, advantages sometimes outweigh disadvantages – sometimes they do not. We will refine models of actor domains and controlled domains, respectively, in either of the styles, component-oriented or object-oriented. Sometimes the expressiveness of object-oriented modeling is appropriate, especially when the whole model can be meaningful placed in a single actor's thread of control. If architectural reasoning is more important, the decomposition in a component-oriented fashion is favored.

3.2.4 Two-Sided Control-Oriented Communication

In *two-sided control-oriented* communication, both parties of the communication relationship maintain a state model of the opposite side in order to exert control on the other party. The situation is informally visualized in figure 3.10.

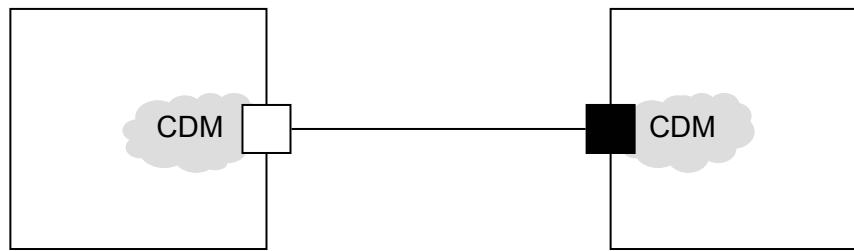


Figure 3.10: Two-sided control-oriented communication

What could be an example for two-sided control, which is not simply composed of two one-sided control relations? A very good example is that of a protocol as we know it in data and telecommunications. Let us take the Transmission Control Protocol (TCP) [Pos81b] as an example: The protocol messages of TCP are exchanged by two parties according to precisely defined rendezvous of message sequences. Each party tracks the current state of the message exchange, assumes the state of the opposite side and reacts accordingly. The relation is not control-oriented in the sense “I want you to do that” but based on collaboration in the sense of “according to our commonly shared plan of action I do the following to achieve our common goal; I assume your cooperation”. For TCP, the common goal is to reliably transfer data from one side to the other under the presumption of disturbances that might hinder transmission of individual messages and/or corrupt data.

A typical characteristic of this type of communication is that the CDM is shared by both parties. This shared CDM is their contract of collaboration and cooperation, which is state-based; otherwise we would not talk of two-sided, mutual control. Alternatively, we call this contract *protocol* and the type of communication *protocol-oriented*.

With this naming convention there is a need to clarify use and understanding of the word “protocol”. Unfortunately, the term “protocol” is and always has been overloaded with different meanings. As we agreed upon in chapter 2, a *protocol* describes valid sequences of messages exchanged between endpoints including rules of message exchange. Although sequence diagrams are a common way to describe certain exchange scenarios (see e.g. [ITU99c]), a full specification of a protocol is often easier to describe by means of a state diagram. The point is that such a protocol specification is just a specification of the externally observable exchange of messages, which does not disclose matters of control. Let us take a simple example: We specify a protocol saying that message a (outgoing) follows message b (incoming) and that b follows a. Who controls whom? Impossible to say unless we know how both parties make use of the protocol.

Most protocols used in telecommunications add a further dimension: one usu-

ally assumes that the message transfer is not 100% reliable. That requires both parties in a communication conversation to carefully keep track of their own and the other party's assumed state. It is the basis for rules like the following: "On my message a my communication partner immediately replies with message b; this is our mutually agreed plan of action. Since I have not received a reply for about 20 seconds, I guess that my message got lost. I will try again and resend message a."² This sort of rules make up most of the CDM and help fulfill the protocol specification (at least to a certain degree of probability). Without such a CDM on both sides, message loss would either freeze communication (waiting endlessly for the outstanding reply) or cause violations of the protocol specification (ignoring outstanding replies).

In addition to message sequences, telecommunication engineers often have this supervising control behavior in mind when they refer to a communication protocol. This understanding of a protocol is not quite in alignment with our definition. However, our investigation on control let us find a precise attributing terminology: Any communication between endpoints can be specified by a protocol regardless of the type of the communication relation. If one-sided control underlies the communication relation, we call the protocol *control-oriented*; if two-sided control underlies the communication relation (as is the case for many telecommunication protocols that must face unreliability, timing issues etc.), we call the protocol *protocol-oriented*. Another case we have not yet touched upon, zero-sided control, calls for the attribute *data-oriented*. As an alternative, we sometimes say for short *control protocol*, *protocol protocol*, and *data protocol*.

One could argue that one-sided control over an unreliable medium requires primarily a protocol protocol that hides the control protocol underneath; a case, which is not uncovered according to our approach. This criticism is absolutely correct and in fact, telecommunications is much about tunneling one protocol under the cover of another protocol. Layering is the design principle to mention in that context, which aims at separating concerns and is the technique to split communication types and protocols from another. Together with layering, it makes sense to distinguish communication types in their pure forms. More about layering can be found in chapter 5.

Note that all remarks about the options how to model a CDM, explicitly or implicitly, component-oriented or object-oriented, reflective or synchronized, are still valid for shared CDMs. There is nothing to add here. Since we will discuss the protocol-oriented style of communication in a telecommunication system in much more detail in another section, we would like to close the discussion here and point the reader to section 3.3.

²This is roughly the functioning of the Alternating Bit Protocol (ABP); it was first described by BARTLETT et al. in [BSW69].

3.2.5 Zero-Sided Control-Oriented Communication

If no side in a communication relationship requires a model of the state space of its communication partner, we say the communication relation is *zero-sided control-oriented*. A better term for that is to say that the communication relation is *data-oriented*.

If no CDM is needed, the communication follows simple patterns of interaction. An example might be the request to multiply the numbers provided in a message and return the result to the caller. One might argue that keeping the callee busy with number crunching is also a way to exert control. That is undoubtedly true, but the point is if this is the caller's intention – and that is something we can barely determine from an external standpoint. Again, we need grey-box knowledge of the caller. We need to look inside that part of the caller's model that unveils its motivation of using the callee's multiply service. If we see that the caller has a semantic state of “keep callee busy”, we know that the communication interface has been misused and transformed into control-oriented communication by the caller. While this seems to be an odd case, this sort of misuse is sometimes intentionally applied to attack computers and offload them. One well-known example are Denial of Service (DoS) attacks. The goal is to cripple the target of attack by an exhaustive number of service requests.

Another example of data-oriented communication is the protocol of the photoelectric barrier of the railroad model, see figure 3.1. The photoelectric barrier only notifies the controller about the event that somebody or something has passed it. For control, the barrier lacks a state space representation of the controller.

If there is a need to classify data-oriented communication, we can attribute it with the category of behavior applied. We stick to the naming convention scheme of OSI primitives. Requesting a multiplication operation is data-oriented communication via *request-confirm*. The photoelectric barrier messaging a passed only is data-oriented communication via *notification*.

3.2.6 Annotating Communication Types in ROOM

We regard control as an important architectural aspect; that is why control-oriented communication relationships should be visible in our models. ROOM does not support a notation for that, nor does any other Architecture Description Language (ADL) the author is aware of.

We propose the following notation, see figure 3.11: For a port whose actor exerts control via the protocol specified for this port, we attach a small arrow directly to the port but inside the actor (to emphasis the grey-box nature of the control semantics). We call such an annotated port *control port*. As an option, a control port may refer in its textual representation to the behavioral component,

the actor references and the classes that specify the CDM. A port whose actor explicitly does not exert control is crossed. We call such an annotated port *data port*. The notation is optional. Ports given in the conventional notation are not further specified regarding their control semantics.

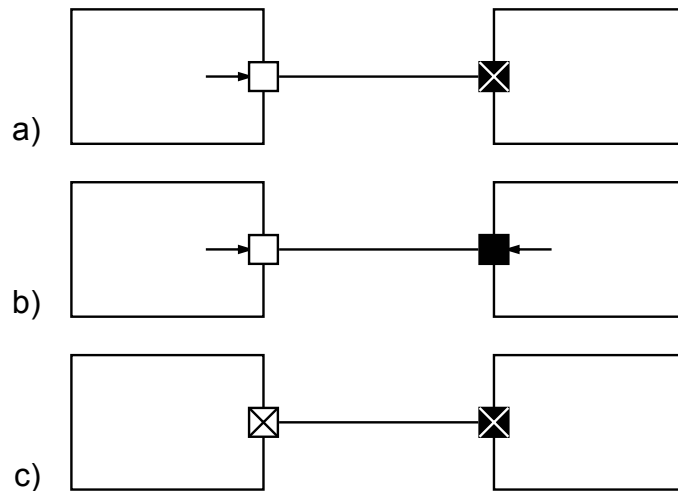


Figure 3.11: Communication types: (a) control-oriented, (b) protocol-oriented, (c) data-oriented

Given the notation for control and data ports, we can easily describe the communication types we discussed previously. One-sided control-oriented communication (or simply control-oriented communication) has a control port on one side and a data port on the other side. We advocate to define the protocol from the data port point of view; consequently, the control port conjugates the protocol. Two-sided control-oriented communication (protocol-oriented communication) has a control port on each side of a communication relationship. Zero-sided control-oriented communication (data-oriented communication) has a data port on both sides.

The value of the notation can be questioned. If a protocol definition combines a control-oriented and a data-oriented communication style, the port cannot meaningfully be classified. Also the target of control cannot always be precisely derived from a ROOM diagram. See for example figure 3.12. Actor A seems to have a control-oriented type of communication with actor B and B seems to have a data-oriented type of communication with actor C – until we get told that actor B just relays all messages from left to right and right to left. So, in fact actor A controls actor C and not B.

The value of the notation is more methodological. First, the notation makes us aware that all communication relations can be composed of the three types we identified. Second, we should aim to model systems in such a way that we

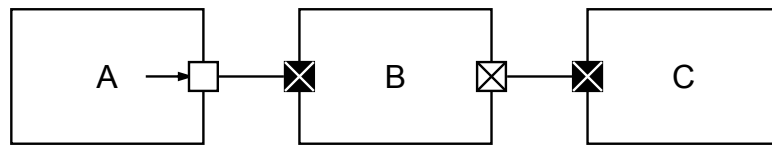


Figure 3.12: Example of control-oriented communication via a relay actor

can clearly separate control-oriented communication from protocol-oriented and data-oriented communication. To achieve this, one can use a powerful modeling technique like layering, which is the main topic of chapter 5. With layering one can either refine or abstract communication relations. For example in figure 3.12, layering could be used to abstract away the relay actor, actor B, so that the target of control becomes clear.

3.3 Communication Services in Telecommunication Systems

According to our generalization of the OSI and TCP/IP reference model, we identified two alignments of communication, horizontal and vertical, and two sorts of communication services, connection-oriented and connectionless. So far, we saw that services are a matter of vertical communication: a *service provider* provides a communication service to a *service user*. The impact on horizontal communication was nebulous, we just learned that horizontal communication is virtual – whatever that means.

We have to delay the study of the precise notion of virtual horizontal communication and its relation to vertical communication. Here, we are interested in the specifics of horizontal and vertical interfaces. How do they look like? How do the standards describe the interfaces, and how can we model them? Is there anything we can say about the types of communication?

To be concrete, we will exemplify the investigation of horizontal and vertical communication on two of the most popular protocols of the Internet: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP provides a connection-oriented communication service, UDP provides a connectionless communication service. Technically speaking, both protocols are protocols of the transport layer. Figure 3.13 visually shows the subject of discussion. The grey shaded boxes, the providers, are the ones, which are specified by the standards. For TCP it is RFC 793 [Pos81b], and for UDP it is RFC 768 [Pos80].³ Protocol standards usually describe in very detail the vertical interface towards a service user, the horizontal interface towards a complementary service provider, and the coupling of the two interfaces.

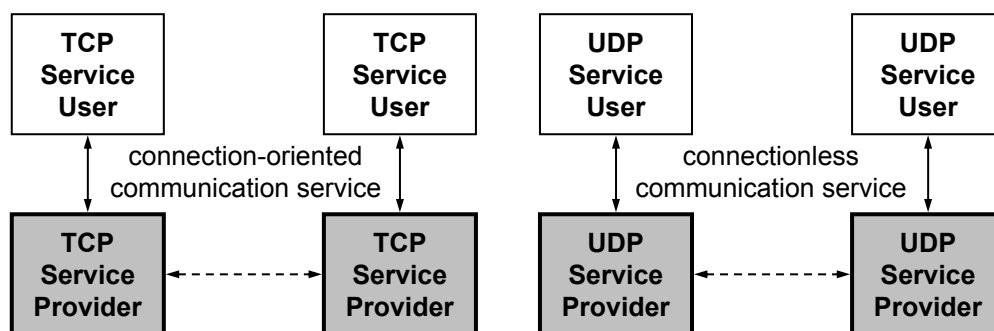


Figure 3.13: Connection-oriented and connectionless communication services exemplified on TCP and UDP

³The standards of the Internet Engineering Task Force (IETF) are called RFCs.

3.3.1 Connection-Oriented Communication exemplified on TCP

TCP provides a reliable end-to-end byte stream over an unreliable internetwork [Tan96]. Essentially, TCP provides a connection-oriented communication service, which has been designed to be robust against many kind of failures.

The TCP RFC specifies all service primitives of TCP as function calls.⁴ We listed the set of primitives already in chapter 2, see table 2.2 (page 38). Since in ROOM we deal with messages between actors, the first challenge is to define equivalent messages for the function calls. Second, we analyze the constituting parts of TCP's service interface, meaning we identify, which part of the interface specification is control-oriented, which protocol-oriented, which data-oriented. Since there is a control-oriented aspect, we will then model the CDM in an object-oriented style. Third is the study of the horizontal interface, the actual protocol that TCP specifies.

Definition of Service Messages

Placed inside an actor, the interface specified by TCP is not accessible in its native form. TCP can be accessed via a ROOM port only, which requires us to define a service protocol i.e. a set of messages for the port. Therefore, we have to redefine the native TCP interface as specified in the RFC (which is specified in form of function calls) into an appropriate message-based interface.

The conversion of function calls into messages is straight and almost trivial: we just take over the function call name as a message name and pack the function call parameters in a data object for message transfer. If a return value is specified for a function call, we agree upon the convention to speak of the function call as the *request* and the retrieval of the return value as the *reply*. We attach a "reply" tag to the message name, in order to distinguish the direction of flow. Note that the request/reply pairs of messages require synchronous execution. However, there may be exceptions from this heuristic. Take the RECEIVE primitive, which does not trigger a *receive* message waiting for a reply, but fetches the values, which have been transmitted by a *receive* prior to the RECEIVE call. This conversion scheme is shown in table 3.1. Note that we renamed "active" OPEN to CONNECT. For the sake of brevity, we did not list the parameters.

In general, the modeller may choose any other suitable convention for a conversion scheme. We recommend to go for name identity or similarity. By default, ROOM converts any incoming message to an event with the event name being identical to the message name. The data object that comes with the message is passed as a parameter to the event. This way, ROOM provides a very cheap and

⁴The function call style is a characteristic of IETF specifications; ITU specifications prefer the message style.

Table 3.1: Mapping of TCP service primitives to service messages

Primitive	Message
LISTEN	listen listenReply
CONNECT	connect connectReply
SEND	send
RECEIVE	receive
CLOSE	close
STATUS	status statusReply
ABORT	abort

easy message/event conversion service. This makes it rather trivial to invoke TCP services via messages instead of function calls. ROOM's conversion feature can also be elegantly used to implement callbacks for event driven applications.

If ROOM's default message/event conversion behavior is not regarded as appropriate or sufficient, one has to design a converter on top of ROOM's default behavior. Depending on the functionality of the converter, this might turn out to be a challenging task. While simple mappings are easy, the implementation of generic functions like known e.g. in CLOS (Common Lisp Object-oriented System) [Gra95] is hard to realize. However, the effort might be worth to turn ROOM into a multiple-dispatched language, because it fosters a much better integration with low-level protocol message decoding schemes. For the physical and the data link layer, sometimes even for higher layer protocols, message compression is commonly used and needs en- and decoding before the message can be interpreted. Message encryption may require such a complex converter function, too. For high-level protocols such as TCP, complex converter functions are rather the exemption than the casual case.

Identification of Service Communication Types

There are seven primitives altogether. Five primitives are concerned with connection handling: LISTEN and CONNECT set up a connection, STATUS supervises the connection, and CLOSE and ABORT shut down the connection. All these commands are about control. They are instructions targeted towards the service provider. They enable a user to request a resource, namely a connection, use it, and release it afterwards. The actual use of the connection is given by SEND and

RECEIVE. This is a data-oriented type of communication embedded in a control-oriented context of a connection.

Consequently, we split up the messages in two sets and define two message schemata: a control-oriented message schema, and a data-oriented message schema, see table 3.2. This implies that the user and the provider of the resource need to have a control port and a data port each.

Table 3.2: Service messages of TCP separated according to communication types

Control Message Schema	Data Message Schema
listen	send
listenReply	receive
connect	
connectReply	
close	
status	
statusReply	
abort	

A reader not familiar with telecommunications may wonder about such a subtle distinction. The separation of control and data is manifested in the architecture design of modern telecommunication systems: systems are sliced in a control and a user (data) plane (see also the comments in the introduction chapter). So far, this should come to no surprise. Novel is, that this separation is applied even on a level of individual interfaces. As will be shown in a later chapter, it is this distinction, the discrimination of control flow and data flow, that lays the basis for a correct, unbiased treatment of horizontal communication. As we discussed previously, horizontal communication is typically subsumed under vertical communication. More on that in chapter 4.

Model the Service CDM

Here, we prefer to develop an object-oriented model of the controlled domain. The CDM can be re-engineered from analyzing the service primitive specification; the semantic information is contained in the parameters of the service primitives.

Several primitives in TCP return or use the *local connection name* as a parameter. The name is used as an identifier for the connection. This is the typical function-oriented way of dealing with objects: they can be addressed only via identifiers inside a function call. In object-orientation, the object can be the container of related functions (called methods) and thereby specifies the context of

the functions in which they operate. Here, we can define a Connection object and attach the following methods to it: `listen()`, `connect()`, `send()`, `receive()`, `close()`, `status()` and `abort()`. The methods are just adapted function calls. A Connection object gets instantiated by the behavior component and is returned as a return value. Then the Connection object is asked either to set up a connection (`connect()`) or to wait for the provider's initiative to actually establish a connection (`listen()`). In addition to the Connection object we identified a further object, a Buffer object. To ease buffer handling, we delegate byte counting and buffer space reservation to the Buffer class. For the sake of brevity, we do not specify buffer methods in figure 3.14.

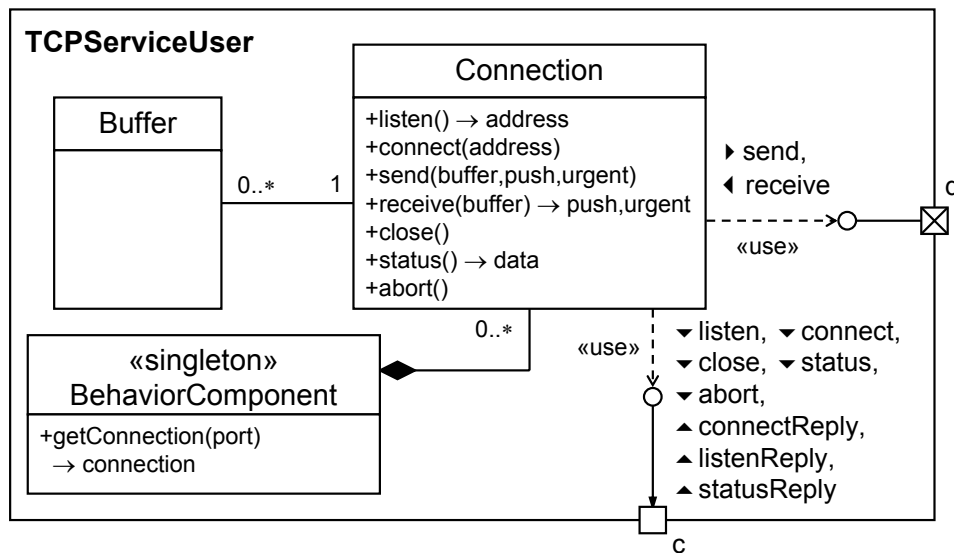


Figure 3.14: Actor model of a TCP service user, object-oriented

Looking at figure 3.14, the task of a Connection object basically is to provide an object-oriented interface to users inside the hosting actor TCPServiceUser and hide message communication. The object functions mostly as a converter and turns method calls into appropriate message calls. A Connection object can be created by the behavior component of the actor (depicted as a singleton class in the diagram); the parameter required to instantiate a connection is a *port number*. This port parameter has nothing to do with a ROOM port; a port number in TCP is an address to discriminate several TCP users and connections, respectively. A lot of port numbers have been reserved for specific TCP user applications by the Internet Assigned Number Authority (IANA).⁵ For example, port number 80 is the standardized address to discriminate the Hypertext Transfer Protocol

⁵See <http://www.iana.org>.

(HTTP) [FGM⁺99] as a TCP service user. An in-depth discussion about addressing can be found in chapter 4.

The BehaviorComponent and the actor TCPServiceUser, respectively, can have zero or more Connections. Each Connection object uses port, c and d, to control the connection and to transfer data. To send and receive a byte stream over port d, a Connection object uses a Buffer object to pass over or to retrieve a sequence of bytes.

Note that the Connection object is very close to the concept of a socket interface object. See for example the two tiny python scripts listed in chapter 2, page 39. Users, who prefer sockets as the de facto standard interface for TCP, may easily wrap the Connection class accordingly.

To recap: Why did we specify a model for the service user and not for the service provider? On an architecture level we are not primarily concerned about the implementation of a service. This may become an issue if we would like to execute our models; usually, a very simplistic implementation that neglects details and works with optimistic assumptions is sufficient for such a purpose. We will describe such a simplified version of a TCP service provider in a subsequent section; so, the service provider model is not out of scope. What counts on an architecture level is the view we are having of TCP from a user's standpoint. The view can be re-engineered from the service protocol as specified in the standard. If this viewpoint is control-oriented then we agreed on to explicate the CDM and model it either in a component-oriented or object-oriented fashion. For a TCP service user, TCP is reduced to the viewpoint of a connection object – and this is a very drastic abstraction of TCP. A user is not concerned about *how* TCP actually realizes the connection towards another party. The user just uses the connection-oriented communication service provided by TCP and works with the abstraction of a connection. To exert control, the service user must have some sort of representation of the world it controls. The notion of a connection is one possible “materialization” of the controlled world, it is an interna of the service user actor; it may be made explicit as we did or it may be implicit and interwoven in the actor's domain model. Since we aim for explicitly, we publish that part of the service user actor and favor a grey-box specification style. If feasible, we draw the “grey internals” on the actor's border to highlight that aspects. Figure 3.15 shows our status quo of modeling the service user and provider part of TCP.

For the sake of a compact notation, the ports used by the Connection class have been attached directly to it. To indicate Connection class multiplicity, the “shadow” symbol has been used; the replication factor of port c and d can be set to one. Remember, this style of a grey-box ROOM actor diagram is informal and just attempts to visualize the information that is textually specified along with the control port. Here, the control port points to the Connection class as the CDM.

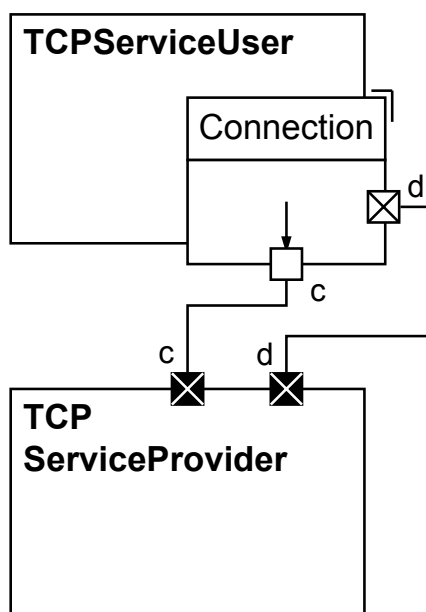


Figure 3.15: Model of the relation TCP service user/service provider

Model the Protocol CDM

The horizontal communication protocol specified by TCP is the main subject of the standard. Roughly 80% of the standard are an in detail description of “horizontal” messages, its parameters and allowed sequences of message exchange. Since a reliable byte stream is TCP’s main concern, TCP undertakes quite some efforts to recognize message loss (messages contain segments of the byte stream), to re-transmit messages, and to recover from disturbances. This complexity is reflected in the format of TCP messages, see figure 3.16. TCP does not specify names to distinguish messages but uses control bits of the flag field (UAPRSF). That is why we define a generic name for all incoming and outgoing TCP messages in our ROOM model, say TCPMessage, and define a data class that specifies the TCP message format as shown in figure 3.16.

A short description, taken from the standard [Pos81b, p.9 f.],⁶ gives a rough overview on the meaning of the fields of a TCP segment for reliable communication. For more details about the message format, the reader is requested to consult the RFC.

A stream of *data* sent on a TCP connection is delivered reliably and in order [from a *source port*] at the destination [to the *destination port*].

⁶Insertions made by the author are indicated by brackets. Highlighting is also done by the author.

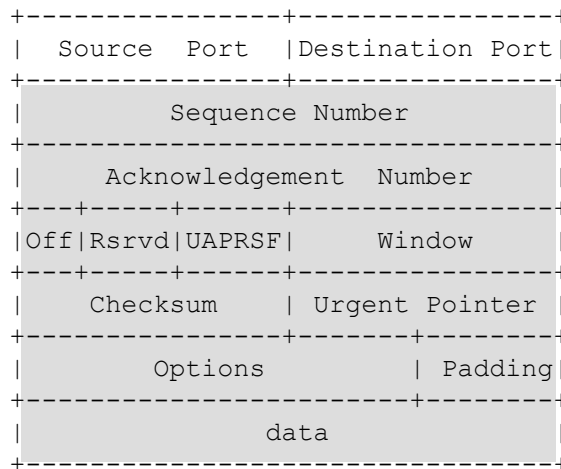


Figure 3.16: Format of a TCP segment, see [Pos81b]

[The data *offset* indicates, where the data begins (since the length of *options* may be variable); field *options* contains – among others – the size of the TCP segment. *Padding* is used “to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros” [Pos81b, p.19]. The *checksum* field is used to recognize segment corruption.]

Transmission is made reliable via the use of *sequence numbers* and acknowledgments. Conceptually, each octet of data is assigned a sequence number. The sequence number of the first octet of data in a segment is transmitted with that segment and is called the segment sequence number. Segments also carry an *acknowledgment number* which is the sequence number of the next expected data octet of transmissions in the reverse direction. When the TCP transmits a segment containing data, it puts a copy on a retransmission queue and starts a timer; when the acknowledgment for that data is received, the segment is deleted from the queue. If the acknowledgment is not received before the timer runs out, the segment is retransmitted.

[...]

To govern the flow of data between TCPs, a flow control mechanism is employed. The receiving TCP reports a “*window*” to the sending TCP. This window specifies the number of octets, starting with the acknowledgment number, that the receiving TCP is currently prepared to receive.

For connection establishment and clearing, TCP uses the control bits to dis-

tinguish phases during the process of setting up and tearing down a connection. Of interest are the *synchronization bit* S(YN), the *acknowledgment bit* A(CK), the *reset bit* R(ST) and the *finish bit* F(IN). With the *urgent bit* U(RG) set, the receiving user should be notified to do urgent data processing; in that case the *urgent pointer* points to the last octet of urgent data. The *push bit* P(SH) indicates data in the TCP segment that must be pushed through to the receiving user.

Looking at how the TCP standard [Pos81b] specifies the protocol unveils a typical problem: It presents the whole TCP service provider by a single state machine and does not clearly separate the TCP protocol from its user (or application) interface. Both are combined, see figure 3.17; it is the result of a white box view.

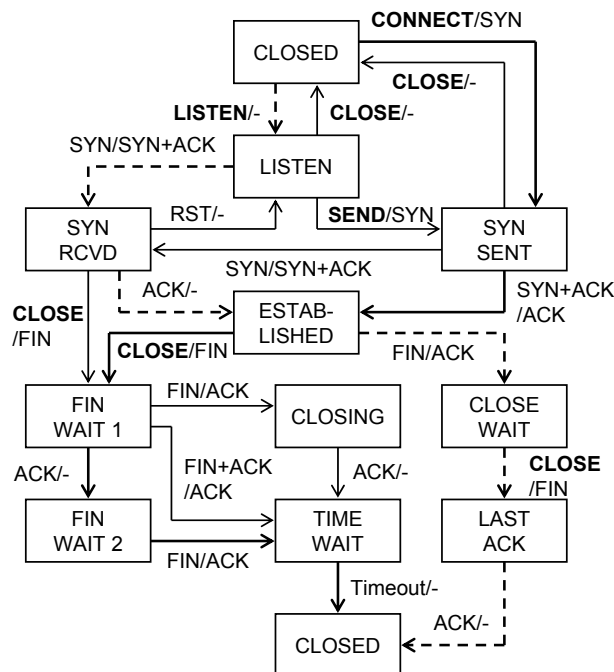


Figure 3.17: The TCP FSM figure is derived from [Tan96, p.532]. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. User commands are given in bold font.

The figure uses a compact notation and shows both the server FSM (Finite State Machine) and the client FSM.⁷ It reads as follows: When a user in his role as a server submits a LISTEN command, the state changes from CLOSED to LISTEN.

⁷Strangely enough, here “client” and “server” denote roles names that indicate the *initiator* and the *acceptor* of a connection request; it says nothing about *how* the connection is used later on. Despite our criticism, people have become so much used to this terminology that we will stick to it.

If, on the other side, the client user submits a `CONNECT`, the TCP protocol sends out a message with the synchronization bit `SYN` set to one, and the client's state changes to `SYN SENT`. On receipt of the TCP message with `SYN` equal to one, the server sends out a TCP message with `SYN` and `ACK` (the acknowledgment bit) set to one and changes to state `SYN RCVD`. When the three-way handshake completes successfully, both parties end up in state `ESTABLISHED` and are ready to send and receive data respectively. This short description of figure 3.17 neglects a lot of details of TCP (e.g. timeouts, which are important to resolve deadlocks and failures) but is sufficient for the purpose of our discussion. The interested reader may consult [Tan96] for more information.

The state machine describing TCP is a good showcase of what is typically understood by the term “protocol” in (tele)communications. It is a specific way of encoding and packaging information, and the exchange of information is precisely described by a state machine. It is the plan of action of two parties, it is their implementation so to speak, that describes how these two parties collaborate in order to provide a service that allows its users to communicate to each other via the abstraction of a connection. We called this sort of collaboration *protocol-oriented*. Thus, TCP's horizontal protocol is a *protocol-oriented protocol* according to our terminology. Still, it remains unclear whatsoever horizontal communication is and how it could be modeled.

One may object that a data-oriented part could be extracted from the protocol protocol. In state `Established`, TCP seems to be solely in data transfer mode, outside of `Established` it is in handshake mode to set up or to release the connection. While this observation is almost true, it is just half the story. During connection setup or release, data may be still passed along with the control bits. TCP is designed in such a way that data transmission is closely interwoven with control information.

As mentioned, the state machine describing the TCP protocol (see figure 3.17) does not accurately separate the protocol part that specifies the mutual control behavior on the horizontal interface from the part that concerns the vertical service interface. There are techniques available that support such a clear distinction as, for example, described in chapter 6. Since we do not want to overload this section with techniques of specification, we assume that such a know-how has been applied so that we can publish just that fraction of TCP that reflects the CDM associated with port `t`, see figure 3.18. The TCP protocol is encapsulated in an actor, `Protocol`, and becomes part of the grey-box specification of the TCP service provider.

Summary

Figure 3.18 summarizes our attempt of modeling a connection-oriented communication service by just taking into consideration the insights we gained about the distinction in communication types. We split the vertical interface into a control-oriented and a data-oriented part and introduced the abstraction of a connection on the service user side. Horizontally, the interface is protocol-oriented. The TCP standard describes how the service provided at the vertical interface is realized by messages sent over the horizontal interface; this description is given by the RFC in form of a single state machine and accompanying text. Due to the peculiarities regarding the relation of the horizontal interface, we use a dashed line for the binding of the *t* ports.

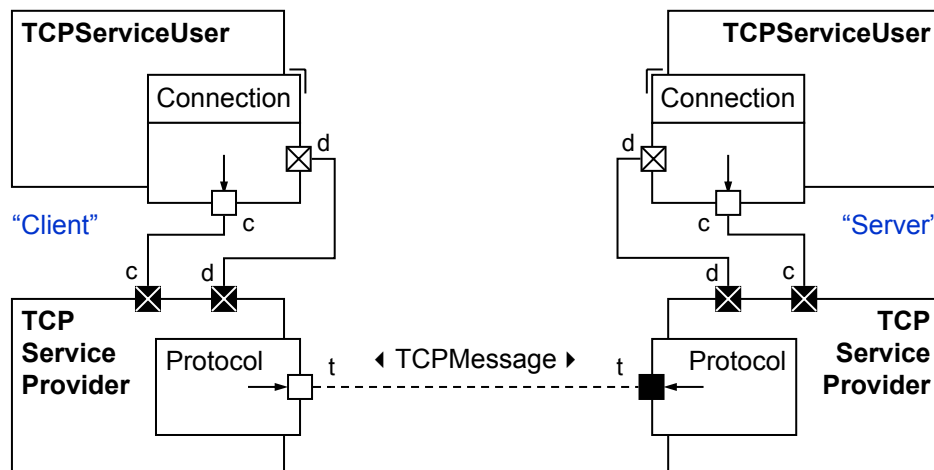


Figure 3.18: Model of a connection-oriented communication service

Since TCP has been just used as an example, we can take figure 3.18 as a generic model for provisioning a connection-oriented communication service in a communication network. Of course, the (implementation) model of the service provider may look different, the details of the *Connection* class inside the service user may vary, but the same arrangement applies.

Note: In the generalized model of a connection-oriented communication service the horizontal interface can be data-oriented. It depends very much, how the standard specifies service realization. Ordinarily, the horizontal interface is protocol-oriented. Regard this note as a hint; we will elaborate on that when we complete our modeling approach.

3.3.2 Connectionless Communication exemplified on UDP

The User Datagram Protocol (UDP) provides a communication service to its users to send messages to other users with a minimum of protocol overhead. UDP is transaction oriented, i.e. it is connectionless, and delivery and duplicate protection are not guaranteed [Pos80].

The service interface for connectionless protocols is very simple; it consists of a send primitive and a receive primitive only. For UDP the service primitive specification is shown in table 3.3.

Table 3.3: Service primitives of UDP

Primitive	Parameters	Return Value
SEND	destination address, buffer address, byte count	
RECEIVE	buffer address, byte count	byte count, source address

If we compare this table with the service primitives of TCP (table 2.2, page 38), we observe that the connection context has been removed from the SEND / RECEIVE part. Hence, each message to be sent out must be tagged with the address of the destination the message is targeted to. The receiver, on the other side, gets the sender's address together with the data received. Without the source address, the receiver might not reply to the originator of the message.

Seen from this perspective, the notion of a connection breaks down to an addressing scheme. Before sending or receiving data it is agreed upon to associate a source/destination address pair with another identifier, called *local connection name* in table 2.2. Any SEND or RECEIVE requires now the connection identifier as an implicit addressing mechanism.

This is a very abstract interpretation of a connection; it does not imply any sort of a line media, physical or virtual, that interconnects communicating parties for message transfer. But we would like to point out that this abstract understanding of a connection is a valid interpretation from a service user standpoint. It is the user's viewpoint (the user's CDM) of the service provider; it is a completely different story *how* the service provider actually implements the service.

Definition of Service Messages

The conversion of the service primitives to service messages is straight and comes without further comments, see table 3.4. Since UDP looks like TCP without the

connection context, it is an easy exercise. Again, for the sake of brevity, parameters have been omitted in table 3.4.

Table 3.4: Mapping of UDP service primitives to service messages

Primitive	Message
SEND	send
RECEIVE	receive

Identification of Service Communication Type

Besides the different addressing mechanisms applied, there is no difference between the SEND / RECEIVE primitives of TCP and the SEND / RECEIVE primitives of UDP. Consistently, the vertical communication type is data-oriented. As a consequence thereof, there is no CDM to model. SEND and RECEIVE are simple operations that submit and fetch data, respectively. There is no aspect of control on a connectionless service interface.

We may provide a method-like interface instead of a message-based interface inside the service user actor as we did for TCP. Here, we leave the decision up to the modeller and do not make any decisions on that. The design of the user internals are of secondary interest, since due to a lack of a CDM there is no need of a grey-box specification style and publish internals to the outside.

Model the Protocol

Connectionless communication services that guarantee neither reliability nor delivery are usually not in need of a protocol-oriented communication type on the horizontal interface. The data handed over via the vertical interface is packaged or extracted from the UDP message format, see figure 3.19, and the UDP messages are just sent out or received on the horizontal interface. That is data-oriented communication.

In its header, each user datagram contains the *source port* and the *destination port*, the *length* of the message and a *checksum*. Following the header, bytes of data fill up the datagram.

Analogously to TCP, we define a generic name for all incoming and outgoing UDP messages, say `UDPMessage`, and define a data class that specifies the UDP message format as shown in figure 3.19.

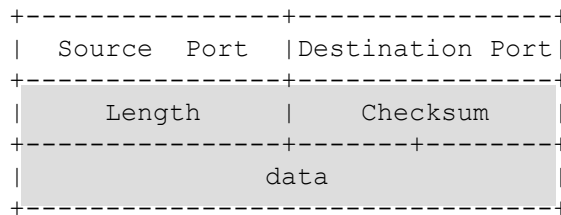


Figure 3.19: Format of a UDP datagram, see [Pos80]

Summary

Figure 3.20 summarizes our attempt of modeling a connectionless communication service by just taking into consideration the insights we gained about the distinction in communication types. The result is remarkable simple compared to connection-oriented services. The vertical interface is of type data-oriented, the horizontal interface as well. As a consequence, there are no CDM's to model: neither the user has to have a viewpoint model of the service provider, nor is there any protocol FSM to model. The relation between the vertical and the horizontal interface from a provider's standpoint is plain and straight, a simple mapping of messages suffices. This simplicity is also reflected in the size of the respective standards: UDP is a document of three(!) pages, TCP sums up to 85 pages.

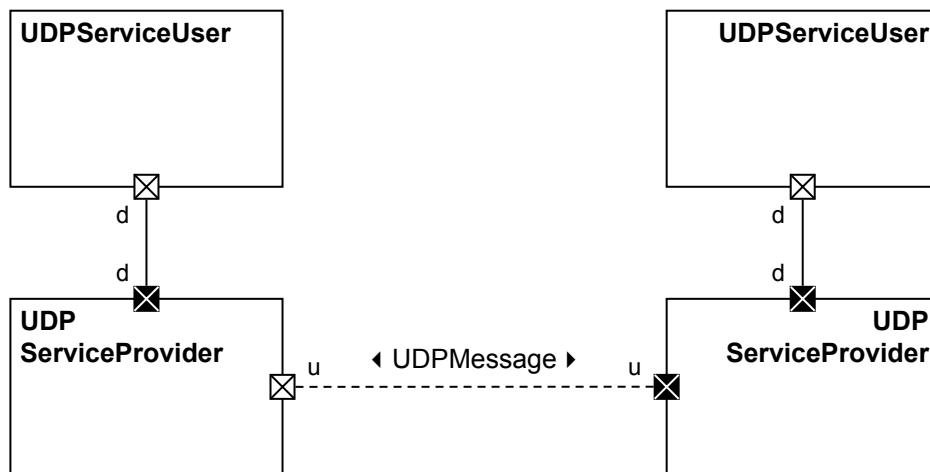


Figure 3.20: Model of a connectionless communication service

Since UDP has been just used as an example, we can take figure 3.20 as a generic model for provisioning a connectionless communication service in a communication network. The same note like for TCP applies.

Note: In principle, the horizontal interface of a connectionless communication service can be protocol-oriented as well. It is an issue of how the standard specifies

service realization. Ordinarily, the horizontal interface is data-oriented. Regard this note as a hint; we will elaborate on that when we complete our modeling approach.

3.4 Resource Control in Telecommunication Systems

A field that is largely ignored in computer networks is the issue of resource control. The term *resource* does not only include physical resources such as adaptors, switchboards, echo cancellers, codec converters etc. but also resources implemented in software. On a software level, resources can be combined, added by some functionality and offer value added services that make a user believe to access a “new” kind of resource that is more than the sum of its physical components. Take for example an alarm clock and a radio, add a composing layer, and you will get a clock radio. The new feature, that the radio turns on at a certain alarm time, is more than any of the resources could provide in isolation.

Most likely, the subject of resource control has been skipped because resources and their control are either regarded as an internal matter of a device or a host, for which no networking is needed, or they are regarded as a matter of the application level, which does not seem worth special mentioning. As a matter of fact, the most popular textbooks on computer networks and distributed systems do not touch upon the subject at all (see e.g. [Tan03]).

We see a need to pay some special attention to resource control. As was mentioned in the introduction chapter, telecommunication systems are sliced in a control and a user plane; basically, it is the control plane that controls the user plane. In most cases this control relationship breaks down to resource control. The control plane controls resources of the user plane. While the control plane and the user plane may operate as largely independent networks, the combining spots are locations of resource control. Usually, the node hosting the resource brings together the control and the user plane. Traditionally, the aspect of resource control has been a local, internal issue. Often, inside such a node, the border between controlling and controlled behavior is blurred and not absolutely separable. At best, the designers defined an proprietary Application Programming Interface (API) for the resource.

One of the intentions of UMTS has been to clearly separate the control and the user plane and to avoid the blur of the control/user plane inside nodes hosting resources; this is the so-called *architectural split* introduced with UMTS. As a result of that, the telecommunication sector of the International Telecommunication Union (ITU-T) defined a protocol, a control-oriented protocol in our terminology, that describes how a user can control a switching centre. This protocol is called Media Gateway Control Protocol (MGCP), it is specified in H.248 [ITU00] and has been taken over as a standard by IETF as well, see RFC 3015 [CGH⁺00]. With the definition of a protocol and the separation in a resource user and a resource provider all prerequisites are given to aim for physical separation of both roles. In the UMTS architecture, these two roles are logically fulfilled by the Media Gateway Controller (MGC) and the Media Gateway (MG). It is up to a manufacturer

to produce two individual nodes or a single combined node. Important is that the distinction has been made logically.

Now we start to have a much better picture of the SIGTRAN functional model, the case study introduced in chapter 1. The MG hosts a resource, basically a switching centre, that is part of the user plane and is controlled by the MGC, with the controller being part of the control plane. Before we value the notion of planes in architecture network models, we first look closer into MGCP and deduce a Controlled Domain Model (CDM) for the controller.

The MGCP is just one example that control-oriented protocols may have a networking dimension associated to them. The alignment of the communication interface is “vertical”, similar to connection-oriented communication. Connection-oriented communication is, in that sense, just a special case of resource control with the connection being the resource.

3.4.1 Resource Control exemplified on MGCP

Some terminology first, directly from the standard: “The Media Gateway (MG) converts media provided in one type of network to the format required in another type of network. For example, a MG could terminate bearer channels from a switched circuit network ([. . .]) and media streams from a packet network ([. . .]). [. . .] The Media Gateway Controller (MGC) controls the parts of the call state that pertain to connection control for media channels in a MG” [ITU00, p.6]. The Media Gateway Control Protocol (MGCP) is the convention used by the MGC to exert control on the MG.

Model the Resource CDM

Since the protocol and its message format are given by the standard, and since the communication type is almost by definition control-oriented, we go straight to the task of developing the viewpoint model (the CDM) for the MGC. In this special case, we also use the term *resource model* for the CDM.

The two key abstractions the MGC relies to have control of, are the notion of a *context* and the notion of a *termination*, see figure 3.21. A Termination is the source and/or the sink of one or more media streams. Properties and statistics of the termination, including the demand to notify the MGC on certain events, are specified via so-called descriptors, abbreviated as *descs* and *Desc* in the ROOM diagram. The standard defines some default descriptors. For example, the *TerminationState* descriptor delivers a simple state model of the termination;⁸ the *Events* descriptor defines events to be detected by the MG and to be reported to

⁸A termination can be in one of the following states: *test*, *in service* and *out of service*.

the MGC; the Audit descriptor identifies the information desired in audit commands, and so on and so forth. A lot of information about the actual resource is delivered via descriptors. Descriptors are a means (a) to dynamically configure the MG and (b) to enable proprietary resources. By intention, the resource model is generic and highly flexible.

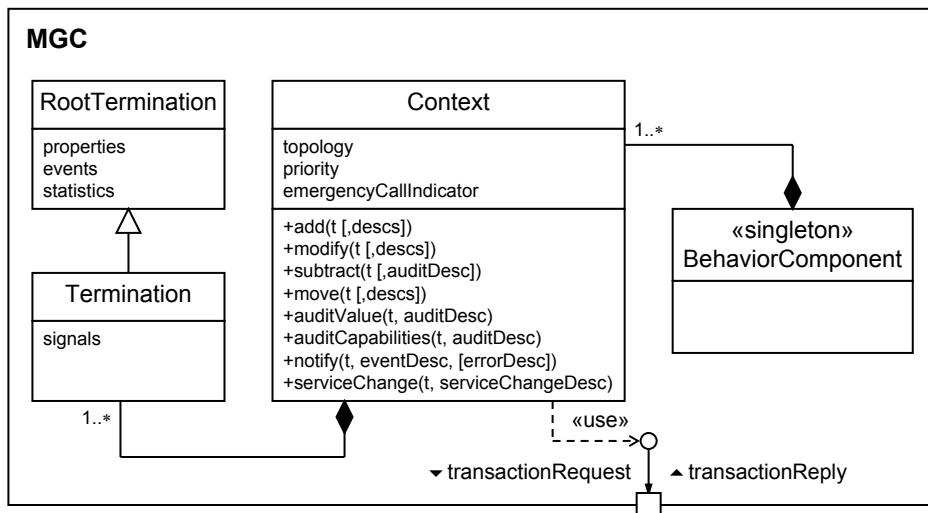


Figure 3.21: Model of the Media Gateway Controller

Terminations can be assembled to a pool of terminations, called Context, and interconnected in a point-to-point or multi-point fashion, which determines the topology of a context. There are commands to `add()` and `subtract()` termination to/from a context and to `move()` terminations from one context to another. Further commands enable the MGC to `modify()` the properties of the termination, to audit terminations (`auditValue()` and `auditCapabilities()`), and to request for notification in case of specific events (`notify()`) and in case of service state changes (`serviceChange`). Some of the commands have to be provided with a descriptor, for others it is optional.

There is always at least a single context, the *root context*, which contains all terminations that are not associated to some other context. If a command must refer to the entire MG and address all terminations, it uses the `RootTermination` for that.

The commands that can be issued via `Context` are encoded either binary or textually, they are packaged in groups of context and submitted in a so-called `transactionRequest` to the MG. Any information that is to be reported from the MG to the MGC is also grouped by context and sent to the MGC in a `transactionReply`. The format of the replies follows the conventions of one or more descriptors, e.g. an error descriptor. Note that the `transactionReply` is processed by the behavior

component first and – if designed like that – handed over to a Context object.

The design of the resource model can be deduced from studying the protocol messages of MGCP. Very often, the basic conceptions are already introduced and explained in the standard. Another resource that may be helpful in re-engineering the resource model from the protocol specification is to consult the Management Information Base (MIB), provided that it exists. The MIB is used for network management purposes. For MGCP a draft version of its MIB is available, see [HABP02].

Summary

Figure 3.22 shows the complete model of the resource user, the MGC, and the resource provider, the MG. Of course, there is more to the MGC and the MG than is shown in the diagram, especially there are some more ROOM ports to be shown that process control information on the MGC level and user payload data on the MG level. Figure 3.22 shows only the fraction that is of relevance for the aspects we are interested here.

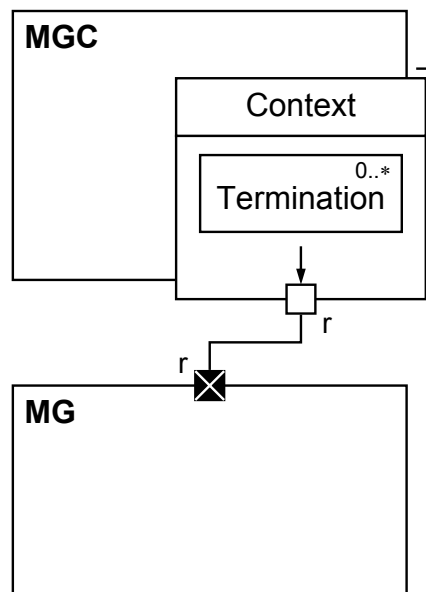


Figure 3.22: Model of a resource control relation

The resource model is exhibited via the informal technique we used before. Drawing class Termination inside class Context is another way of visualizing containment in UML.

Figure 3.22 and the introductory railroad example have very much in common since both represent control-oriented communication in its plain form. There is

a clear relationship of who controls whom. In both cases, the resource model is essential in order to gain and exert control. The resource provider has no state space representation of the controller. The events the MG is instructed to notify the MGC about correspond to the signals emitted by the photoelectric barrier. MGCP is just a telecommunication specific example of a control protocol.

3.5 Summary

In this chapter we investigated thoroughly the aspect of control in communication systems. We defined control as the directed exertion of influence on the behavior of some other entity. Having control implies to have a state space representation of the controllee. We called this representation a Controlled Domain Model (CDM), which is a viewpoint of the Domain Model (DM) of the controllee. It is the CDM that we try to make explicit in our models. However, it might be difficult to clearly separate control behavior from the state space tracking of the CDM. Nonetheless, it is even worthwhile to explicate the conceptual entities under control; it makes models more expressive and understandable.

We proposed an optional notational extension to ROOM in order to classify ROOM ports as either control ports or data ports. Control ports are indicated by a small arrow attached to them, data ports are “crossed” ports. Control ports may maintain pointers to that part of the actor’s implementation model, which realizes the CDM. Informally, we bring this grey-box kind of information to the reader’s notice by drawing the corresponding actor references or classes on the actor’s border, see for example figure 3.23.

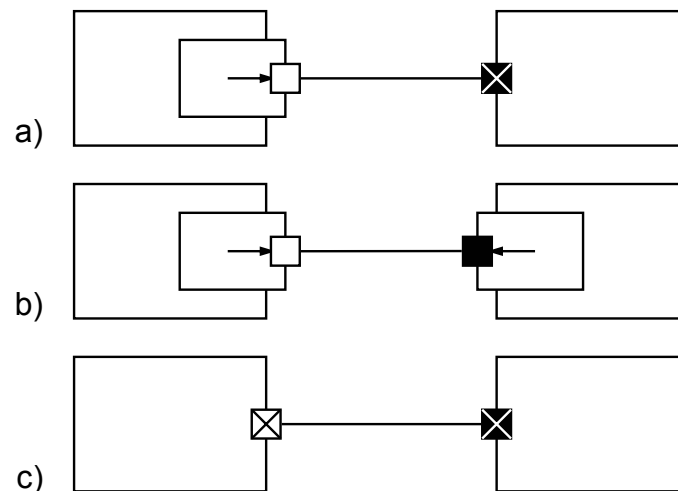


Figure 3.23: Summary of communication types: (a) control-oriented, (b) protocol-oriented, and (c) data-oriented

Given this notation, figure 3.23 summarizes all three identified communication types: (a) one-sided control, (b) two-sided control, and (c) zero-sided control. We agreed on the names control-oriented, protocol-oriented and data-oriented as characterizing attributes. We also mentioned that the notation is foremost of methodological value, since the port types do not necessarily unveil the underlying communication type (e.g. in case of relaying actors).

We then studied which kind of communication types there are in a telecommunication system. Basically, on a coarse granular level, a telecommunication system is composed of communication services (connection-oriented and connectionless) and resource control services. We exemplified the study on a representative selection of three protocols and looked at how to model the vertical and the horizontal interface, respectively. The protocols we examined are TCP for connection-oriented communication, UDP for connectionless communication, and MGCP for resource control.

If we look at the interfaces individually, neglect their coupling, and generalize the example protocols, we get a much better picture of the constituting communication types per interface type. Figure 3.24 summarizes our findings for the vertical interface. The open box symbol for the actors indicates that there may be other aspects (like other interfaces and related CDMs), which we do not show.

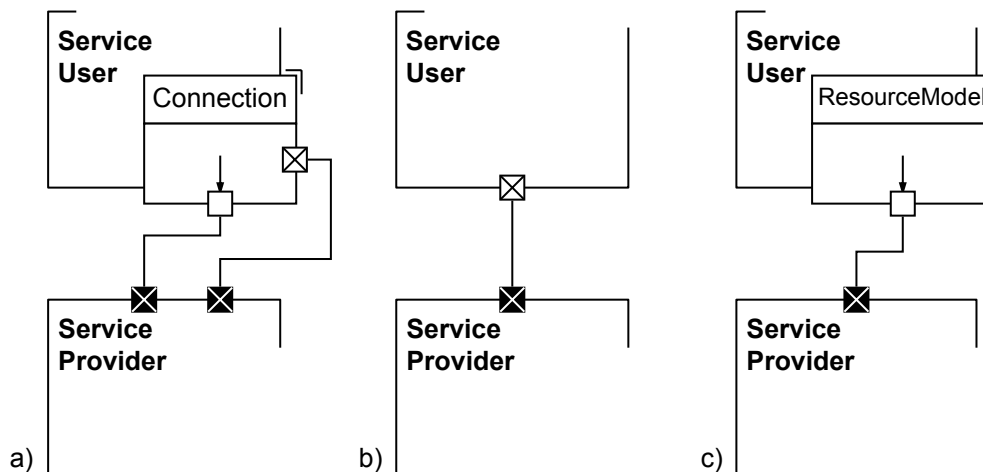


Figure 3.24: Vertical communication types for (a) connection-oriented, (b) connectionless, and (c) resource services

Figure 3.24 a) represents a connection-oriented communication service provided by a service provider to a service user. The service user works with the abstraction of one or more connections. We can clearly separate the aspect of control to set up, supervise and release a connection, and the actual transfer of data. This separation is reflected on the vertical interface, since it is split in two parts. One part is of type control-oriented, the other of type data-oriented. For a connectionless communication service, figure 3.24 b) shows that the vertical interface is of data-oriented communication type only. Therefore, no Controlled Domain Model (CDM) is required on the service user side. For resource control, figure 3.24 c), the service user maintains a resource model and has a control-oriented type of communication towards the service provider.

For the horizontal interface, figure 3.25 summarizes our findings. We are still not clear on what the horizontal interface really is and indicate that by a dashed binding, but the distinction in communication types is nonetheless possible.

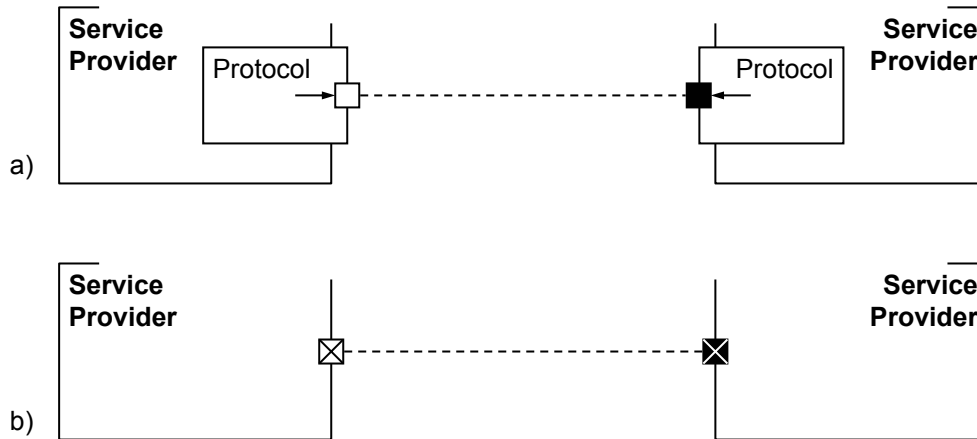


Figure 3.25: Horizontal communication types for (a) a protocol-oriented and (b) a data-oriented protocol

Figure 3.25 a) shows a protocol-oriented communication type. Both sides of the communication relation exert control in order to deal with the deficiencies of message transmission over the (dashed) binding. Consequently, both sides maintain a state model; this model is what telecommunication system engineers often refer to when they use the term “protocol”. We agreed on to call such a protocol a protocol-oriented protocol, or protocol protocol for short. Figure 3.25 b) shows an alternative, a data-oriented communication type. Messages are just released to the horizontal interface, no care is taken if a message reaches its destination or not.

It is interesting to note that the vertical interface is either purely control-oriented or purely data-oriented or a combination thereof; but it is not protocol-oriented. The horizontal interface, on the opposite, is either protocol-oriented or data-oriented but not control-oriented. If this observation scheme holds valid, we have found a first criterion that semantically classifies horizontal and vertical communication by an excluding property: a vertical interface is never protocol-oriented, a horizontal interface is never control-oriented.

Our modeling approach starts to take shape. The protocols mentioned in the case study (chapter 1) fit all to the classification scheme for vertical and horizontal interfaces. SCTP, for example, is a protocol offering connection-oriented communication service and is very similar to TCP. IP and MTP3 offer a connectionless communication service very much the same as UDP does.

The next chapter is about the aspect of distribution, which is primarily a matter of horizontal communication. In that chapter we will eventually reveal what the

dashed binding in previous figures is all about.

Chapter 4

Distribution

The most obvious characteristic of a communication system is its aspect of distribution. If two or more processes, users or – more abstractly – entities are physically spread in space but want to collaborate, they somehow have to bridge spatial distribution and establish communication. This chapter concerns only the aspect of distribution and how to model services that enable entities to communicate. That includes the important area of how to treat addressing.

In section 4.1, we formally discuss the aspect of distribution with the algebraic toolset provided in chapter 2. We then go on and transfer our insights to the modeling language ROOM in section 4.2. In section 4.3 we thoroughly discuss how to model distribution and communication networks and exemplify it on TCP, UDP and JavaSpaces. The notion of addressing, an often neglected area in system architecture modeling, is subject of section 4.4. In section 4.5 we summarize and generalize our considerations and derive the next set of elementary patterns for our modeling approach.

4.1 What is Distribution?

Definitions on distribution to be found in literature suffer preciseness on the one hand and generality on the other hand. We will mathematically define the notion of distribution, which satisfies the demand for precision and generality, before we move on to think about the capability of ROOM to model distribution.

4.1.1 Definitions in Literature

How do we define *distribution*? In literature there is no common agreement on a suitable definition. For example, WU defines distribution as follows [Wu98, p.12]:

A distributed system is one that looks like an ordinary system to its users, but runs on a set of autonomous processing elements (PEs) where each PE has a separate physical memory space and the message transmission delay is not negligible. There is close cooperation among these PEs.

According to his definition WU concludes that computer networks are not considered distributed systems. WU argues that processes at different locations or sites do not work cooperatively. Say, in a network of independently working PCs (Personal Computers) the network is only used to exchange certain information such as email. We do not favor such a restrictive, narrow interpretation of “cooperativeness”. As we will shortly see, it is also a matter of the abstraction level taken. In the extreme case, the users are sort of PEs who cooperate via email using the resources of a network.

Other authors, like TANENBAUM and VAN STEEN, try to give a loose, very liberal characterization only [TvS02, p.2]:

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

Still, the authors make use of a rather technical terminology, they talk of computers or – in case of WU – of processing elements, which unnecessarily restricts the interpretation of the terms.

We think that at an abstract level distribution is primarily a logical conception and that it is adequate to give a formal definition based on a proper model. Secondary, distribution has a technical dimension. That means the logical conception needs to be filled with some technology and with terms of that technology domain. Dependent on the domain it might be appropriate to talk of processing elements, computers, or something else. Key is to catch the nature of distribution underlying all these technologies.

4.1.2 An Algebraic Model of Distribution

As a starting point, let us take a logical entity, a component that encapsulates some functionality. A prerequisite for distribution is that the component under consideration can be logically split up (“decomposed”) into separate parts, each of the parts representing a new component. The parts communicate to each other via precisely defined interfaces in form of interacting messages; the message set exchanged between two interfaces is usually subsumed under the term *protocol*. In other words, a monolithic (meaning non-distributed) entity gets refined by a network of separated but cooperating parts. From an outer perspective, the conglomerate of parts preserves the message syntax and the semantic behavior that can be experienced at the interfaces of the monolithic entity. The “outer” interfaces do not necessarily need to be distributed over the separated parts; that is up to the design rational and the objective target of the distribution.

Assumed that a component L can be refined into n subcomponents L_k ($k \in \{1 \dots n\}$, $n \in \mathbb{N}$, $n \geq 2$), some of them exposing a communication relation. For the sake of brevity we presume *mutual feedback* composition “ \otimes ” only. For simplicity, we assume that for the components L_1, \dots, L_n their sets of channels are pairwise disjoint. To connect the channels we use simple connectors $SC_{i,j}$ with $(i, j) \in N \subseteq \{(i, j) : 1 \leq i < j \leq n\}$, which basically rename the channels. Formally, each component $SC_{i,j}$ renames a set of channels $\{x_1, \dots, x_k\}$ into a set of channels $\{y_1, \dots, y_k\}$. Thus we have

$$\llbracket SC_{i,j} \rrbracket = (\bar{x}_1 = \bar{y}_1 \wedge \dots \wedge \bar{x}_k = \bar{y}_k)$$

The refinement of L is then expressed by the formula

$$L \rightsquigarrow \exists x, y : \bigwedge_{1 \leq i \leq n} \llbracket L_i \rrbracket \wedge \bigwedge_{(i,j) \in N} \llbracket SC_{i,j} \rrbracket \quad (4.1)$$

where x and y are the sets of internal (renamed) channels.

Quite often it is overseen that formula 4.1 just describes a functional decomposition, which is a preparation step for distribution; it is not the actual distribution onto a network of computing entities. The design challenge is to find a suitable functional decomposition that serves as a basis for distribution.

When talking about distribution, we refer to the logical conception that the functional parts get spread over, say, hosts or physical nodes, meaning that they require some sort of communication means in order to bridge the spatial separation. That is an important difference to a purely functional split-up. The interaction of the functional parts in a distribution network is *not* fault-free per se; it is sensitive to disturbances on the communication medium and dependent on the properties of the connection. We condense the whole communication medium in a

model of a connector, which we call *complex connector*. The complex connector is a component that represents the properties of the communication channel and its effects on the transmission of messages. These properties are called Quality of Service (QoS) attributes and include all relevant characteristics, such as *reliability*, *throughput* etc.

There is a large variety of connectors. In the simplest case, a complex connector is just a renaming of channels and the identity on the respective streams. In the most general case it can be a complex relation between its input and output channels.

Given a set of n components L_i , $1 \leq i \leq n$, with pairwise disjoint sets of input and output channels, we may use a set of complex connectors $C_{i,j}$ with $1 \leq i < j \leq n$ and $(i,j) \in N \subseteq \{(i,j) : 1 \leq i < j \leq n\}$. Each pair in N is called a *communication relation*.

We construct a network of the components L_i connected by the complex connectors $C_{i,j}$ by the specification

$$L \rightsquigarrow \exists x : \bigwedge_{1 \leq i \leq n} \llbracket L_i \rrbracket \wedge \bigwedge_{(i,j) \in N} \llbracket C_{i,j} \rrbracket \quad (4.2)$$

where x is the union of all channels in the complex connectors. Again, we assume that for all connectors their sets of channels are pairwise disjoint.

In the simple case, all $C_{i,j}$ are the same after renaming the channels; in the general case, there is a different connector $C_{i,j}$ for each communication relation in N .

As we can see in comparison to equation 4.1, the functional decomposition network is a distributed communication network assuming ideal communication conditions, the complex connectors get replaced by non-time delaying, ideal “short circuits”. Equation 4.1 turns out to be a special case of equation 4.2. That is an important observation. Many popular modeling languages assume ideal communication relations only. Typically, a solid line represents an ideal communication binding between entities. Distribution networks cannot be modeled like that. We need means to distinguish ideal from non-ideal communication, that is we need to distinguish between internal (ideal) communication and external (disturbed) communication to a remote peer. Furthermore, we need to specify the model of the external communication relation. If we do not introduce proper modeling enhancements for that, our models suffer from a loss of semantic expressiveness.

It is important to note that our mathematical definition of distribution requires equation 4.1 and equation 4.2 to be valid; we call this the demand for *global identity*. Equation 4.1 says that some functional parts put together compose some “larger” functionality L ; the behavior of that functionality can be experienced at some “outer” interfaces, the interfaces of L . Equation 4.2 puts a further require-

ments on the design of the functional parts: each part L_k has to be designed so that it compensates the communication deficiencies modeled by the complex connector. The complex connectors $C_{i,j}$, on the other hand, do not have functionality that contributes to the composite L ; that is guaranteed by equation 4.1.

Our definition of distribution, condensed in the two equations above, is compatible with most definitions found in literature. Many definitions like the ones quoted match perfectly with equation 4.1 but most authors are very implicit or use rather cryptic paraphrases for the fact of equation 4.2. WU speaks about non-negligible delay in message transmission between processing element and forgets to mention other non-negligible QoS attributes; TANENBAUM and VAN STEEN rely on the implicit knowledge that independent computers must be somehow connected in order to appear as a single coherent system. The advantage of our formal approach is evident.

4.1.3 The Complex Connector

In our definition of distribution the notions of a *simple connector* SC and a *complex connector* C are most striking. Whereas the simple connector is merely a helping construct that explicitly captures the interconnection of components via its channels, the complex connector adds a new dimension: all properties of the transmission media used to bridge the “gap” between remote entities are condensed in a model of QoS attributes.

Properties like *delay* and *jitter* impact time in a timed stream and are subsumed under QoS attributes. These attributes can be either implicitly included in the model of C or can be explicitly captured by a specification Q in a glass-box model of C . For the sake of clarity, we prefer the latter. Other properties such as probability of *message loss* or *message duplication* [ITU94] also come under QoS attributes and have an impact even on untimed streams.

We denote the part of C that concerns only the aspect of connectivity by SC . SC corresponds to the simple connector introduced along with formula 4.1; it is C without Q .

In the most general case, all input and output of component C has to pass component Q , i.e. $I_Q = O_Q = I_C \cup O_C$, where we assume $I_C \cap O_C = \emptyset$, see figure 4.1. The syntactic interfaces of C and SC are identical: $C \in (I \triangleright O)$ and $SC \in (I \triangleright O)$.

We call this idea of filtering *total filtering* and denote it by the operator \otimes , which is not commutative by stressing the different roles of Q and SC . We say that “ SC is (totally) filtered by Q ”. Formally, we define total filtering as follows

$$Q \otimes SC = Q \otimes SC$$

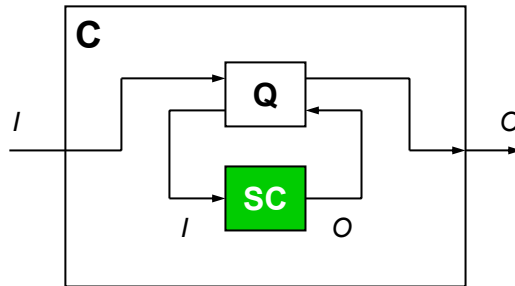


Figure 4.1: Modeling Quality of Service (QoS) attributes

where we assume $Q \in ((I \cup O) \triangleright (O \cup I))$ and $SC \in (I \triangleright O)$. Note that then

$$Q \otimes SC \in (I \triangleright O)$$

In this specific context, the filter models QoS attributes. An elaborated model of a complex connector between two communicating peers explicitly respects QoS attributes and clearly distinguishes between the connectivity function and its QoS aspects:

$$C \rightsquigarrow Q \otimes SC \tag{4.3}$$

Very often, the formulation of QoS attributes for a complex connector is essential in network design.

4.2 Introducing the Complex Connector in ROOM

We are not going to use the algebra to formally specify models of communication but use ROOM instead. Consequently, we uplift ROOM's language capabilities by the notion of the complex connector, a conception which we identified as essential to model distributed systems.

4.2.1 Discussion of Solutions

The notion of a *simple connector* as introduced in formula 4.1 matches perfectly with the notion of a binding in ROOM. Simple connectors connect components, so do bindings via ports. Simple connectors like bindings do not have any impact on the messages they transport, they behave ideal: delivery is immediate without any time delay, the message order is preserved, and the message and its content experience no modification of any sort. Simple connectors and bindings are structural properties in our models, they manifest communication relations. Besides components, simple connectors are first class concepts, neither can be substituted by the other.

For the notion of a *complex connector* as introduced in formula 4.2 there is nothing comparable in ROOM. Basically, there are two options to introduce complex connectors in ROOM models: (1) The complex connector is modeled by a distinct actor class and gets inserted between two communicating actor classes. Still, the actor imitating the complex connector and the communicating actors are connected by ideal bindings. This situation is shown in figure 4.2 a). (2) The concept of a binding is extended. The proposal is to extend a binding by an optional reference to an actor class specifying its behavior. The actor class must have two ports (without replication!) that complement the message schemata (the ROOM protocols) defined for the ports the binding interconnects. This situation is informally shown in figure 4.2 b). The arrangement is very close to the concept of channel substructures in SDL [EHS97, p.121 ff.]. For a more precise definition of the ROOM enhancement watch out for chapter 6.

The advantage of the first option is that there is no need to enhance ROOM. Its major drawback is that the complex connector loses its status as a first class concept. The complex connector becomes an actor among others and is poorly reasoned as an structural concept of semantic importance. We therefore advocate the solution to extend bindings by an actor class reference.

4.2.2 The Extension: Typed Bindings

The language enhancement comes at little cost. The binding remains a first class concept and gets improved by a typing concept: the referenced actor class is in-

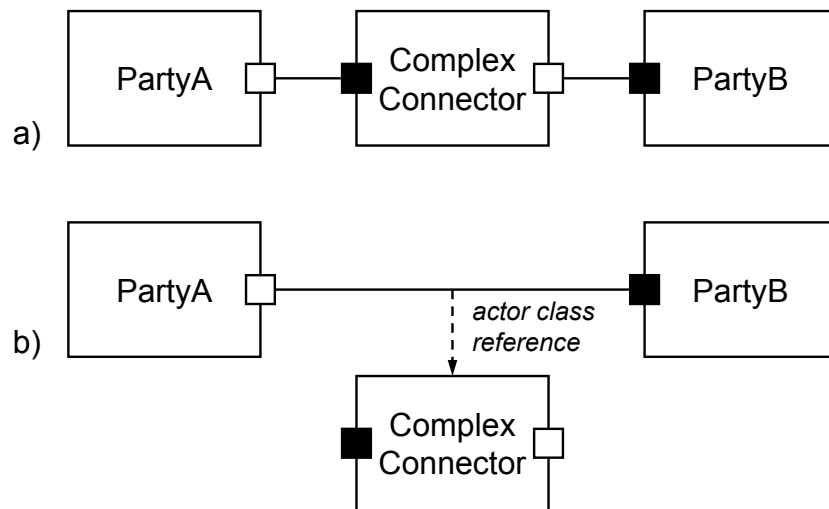


Figure 4.2: Options to introduce complex connectors in ROOM: (a) actor class, (b) enhanced binding concept

terpreted as the binding's type. If actor class A_1 is the parent class of A_2 , and if binding specification B_1 refers to actor class A_1 and B_2 to A_2 , then the type of B_2 is a subtype of B_1 . That means, bindings can be easily exchanged by subtyped bindings in a model.¹ If we define the type of the ideal binding (not explicitly referring to any actor class) to be *ideal*, all other non-ideal bindings are a subtype of ideal.

With the type concept, bindings become sort of detached from their local scope within an actor class. In ROOM, bindings are individually defined for each pair of ports within an actor class, i.e. between ports of actor references or between a port of an actor reference and an endpoint of the containing actor class. If typed, the binding refers to something outside the containing actor class context, namely to another actor class that specifies a message schema for the binding (via the ports of that actor class) and some behavior. That way, several non-ideal bindings may share the same specification; they are said to be of the same type.

By using an actor class as a specifier for bindings, one can take full advantage of all ROOM features to model and decompose a binding. This said, it is even possible to use typed bindings within such an actor class. Since typed bindings are foremost our means to express the aspect of distribution, it may seem – at first sight – puzzling to define a binding that can be itself subject to distribution. As we

¹Strictly speaking that statement can be proven wrong. In ROOM, the child actor class must not be a rigorous specialization of its parent class. This oddity impacts also the type concept for bindings. But the authors of the ROOM book annotate that there have to be good reasons to violate the principle of rigorous specialization [SGW94, p.261]. So, in most cases, the footnoted statement will be true.

will soon see in chapter 5, a binding can also act as an abstraction that resolves to the notion of an intermediate system.

Basic Notational Convention

For typed bindings we introduce a new symbol that we are going to use in our ROOM diagrams: slim to remind of a binding but somewhat boxed to indicate that some complexity in terms of behavior is attached to it, see figure 4.3. The name of the referenced actor class is written inside the symbol and, by convention, starts with a small letter. The ports the typed binding connects must find their (unique) conjugated counterpart at the actor class the binding refers to. The notation suggests an option to indicate bidirectional message flow and unidirectional message flow. If of relevance this addition may be used.

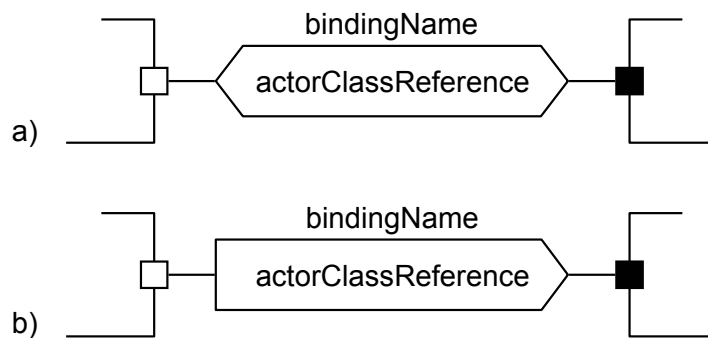


Figure 4.3: Notation for typed binding: (a) bidirectional, (b) unidirectional; binding name is optional

The name “typed binding” refers to our realization of the complex connector concept in ROOM. But note that our definition of a typed binding is not identical to the notion of a complex connector; in fact, we relaxed an important constraint which constitutes the complex connector. It is possible to “abuse” the concept of a typed binding to such an extent that it turns out to be very much alike an actor, but limited to two ports only. If, for example, a typed binding consumes two subsequent messages (each message carrying an integer number), adds their data words, drops the two messages and ejects a new message with the sum of the integers, this would severely violate our definition of a complex connector. Our definition demands that the exchange of all complex connectors by simple connectors (that preserve syntactic interfaces, of course) has no impact on the general functionality of the whole – disregarding time delays. The constraint of *global identity* is not guaranteed by the use of typed bindings. Later, in chapter 5, we will revisit our definition of distribution and reason for local identity instead

of global identity. Until then, we have to validate the correct use of typed bindings on a network-wide scope.

The definition of a complex connector might lead to connector specifications that remind us little of the QoS attributes, which we originally had in mind. Look at figure 4.4: One typed binding refers to an actor class that does message encryption, the other typed bindings do message decryption. This sort of use of a typed binding relates to the idea of factoring out *aspects* to the connector and has been around as an enhancement to connector-based ADLs for some few years [SG96]. Novel is our way of approaching the complex connector including its definition. As odd as it may seem, as long as the actors still function correctly with the typed bindings replaced by ideal bindings, there is nothing to object against such specifications of complex connectors. Rather, we may have to revisit our narrowed scope of understanding. A complex connector is an abstraction of a connection medium, which goes beyond capturing traditional QoS properties. That is to keep in mind, because we will mostly use the complex connector for modeling QoS.

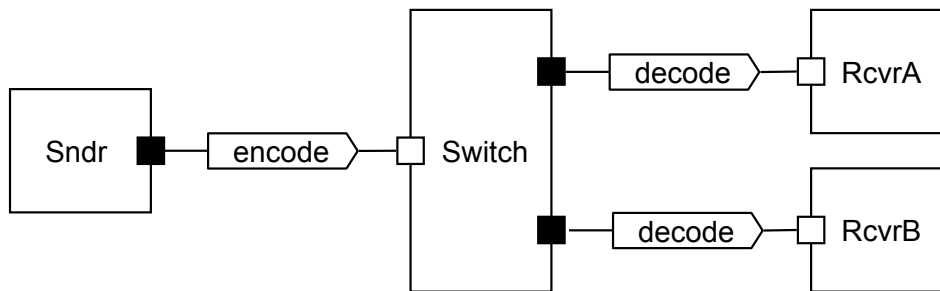


Figure 4.4: Example: En-/decoding complex connectors

Enhanced Notational Convention

References of a typed binding to an actor class correspond very much to the concept of actor references in ROOM. Naturally, one may ask whether there are features related to actor references that may be equally applicable for typed bindings. That is a tricky question because the main difference between actors and bindings is in the incarnation/destruction process: Incarnation and destruction of bindings is tightly bound to the creation and destruction process of actors; bindings depend on actors, they are incarnated and destroyed along with the actors. Actually, establishing a typed binding means to incarnate the referenced actor class and to insert the actor class instance in the line of communication. This is what we call a *fixed* typed binding in analogy to fixed actor references. It is the default incarnation mechanism for typed bindings and corresponds to the casual incarnation process for (ideal) ROOM bindings. An *optional* typed binding is not created along with

the pair of actors it should connect, its creation is suppressed. Optional typed bindings have to be dynamically created and destroyed by command after the actors to connect have been incarnated. The command automatically connects a pair of ports of the specified binding (given by the binding name) that have not been connected yet. Similarly, an *imported* typed binding means to import an compatible actor into the first available slot of the specified communication relation. The notation for fixed, optional and imported typed bindings is shown in figure 4.5.

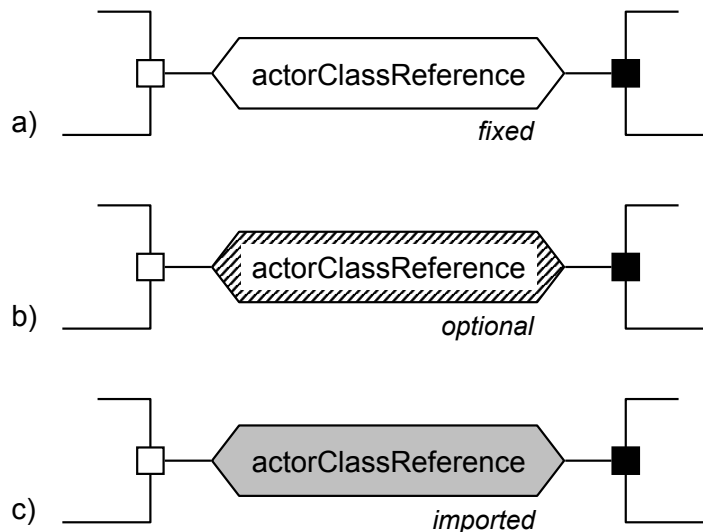


Figure 4.5: Extended notation for typed bindings: (a) fixed, (b) optional, (c) imported

The notation is fully compatible with the notation and the commands for actor references (see also chapter 6). A replication factor, obviously, is of no use for typed bindings. Each referenced actor class must have only two ports without replication. One could also introduce a substitutable reference modifier for typed bindings which we have not due to a lack of practical use in our models. As the reader will notice, in our models we even manage to survive with fixed typed bindings only.

Generalizing the Notion of Bindings

One could think about generalizing the notion of a binding. What about bindings that do not only interconnect two but three, four or more ports? As a matter of fact, there is no reason why not to introduce the generalization of an n-ary complex connector and n-ary typed binding, respectively. But keep in mind that multi-party bindings are not as trivial in design as two-party bindings are and can easily lead to constructs we possibly perceive as counterintuitive. For example, if we

would use the complex connector to model connectionless communication, we would have to give up the understanding of a complex connector as a model for a connection! While methodically consequent and correct, we decided not to use n-ary typed bindings but use actor classes instead, which stand for the respective sort of communication service to be provided. By this approach, we blur the border of a clear mathematical reasoning and a clear methodical approach but become more intuitive from the standpoint of an engineer. This is a typical case, where one has to judge between a stringent approach and its applicability.

In order to not completely blur the border, we decided to use a special notation for components, which are modeled as actor classes but could be (methodically correct) modeled as n-ary typed bindings as well. These actor classes are symbolized as rectangles with rounded edges. ROOM++, our extended version of ROOM, supports n-ary typed bindings.

Remarks

Note that typed bindings can be only specified between ports of actor references! The communication between a relay port of an actor class and the reference port of an actor reference cannot be subject to distribution. An actor class can be the logical container of a set of actor references that make up a network of distributed, remote entities, but it cannot be the border on which inner typed bindings end. If so, we would run into severe problems of keeping a model consistent and composable. See, for instance, figure 4.6. The typed binding b1 imposes implicit requirements on the communication relation of port p1 and p2 (e.g. b1 may rearrange the order of messages) that remain totally hidden for the design of actor class A2. A2 is just designed to compensate transmission defects of b12 (say, on the average each tenth message gets lost) and is messed up by the unexpected wrong order of messages.

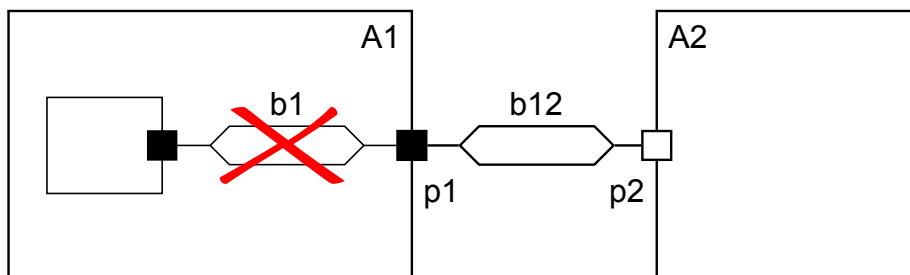


Figure 4.6: Example of an invalid specification with typed bindings

Note also that one has to be extremely careful with the design of actor references that have a typed binding attached to them. These actors may undermine the

aspect of distribution by tunnelling communication via the behavior component. Similarly critical is the use of ROOM SAPs. If not carefully used we may mess up our models of distribution. Distributable entities should be modeled as independent as possible having as little contextual dependencies (modeled by ROOM SAPs and the behavior component) as possible. Although our models are only logical models of distribution, they should be deployable in practice.

4.2.3 The Algebraic Definition Visualized

With the new notation introduced we can visualize the algebraic definition of distribution. Figure 4.7 a) shows the functional breakup of an actor L into several composing actors. Figure 4.7 b) shows the replacement of ideal bindings by typed bindings modeling message loss.

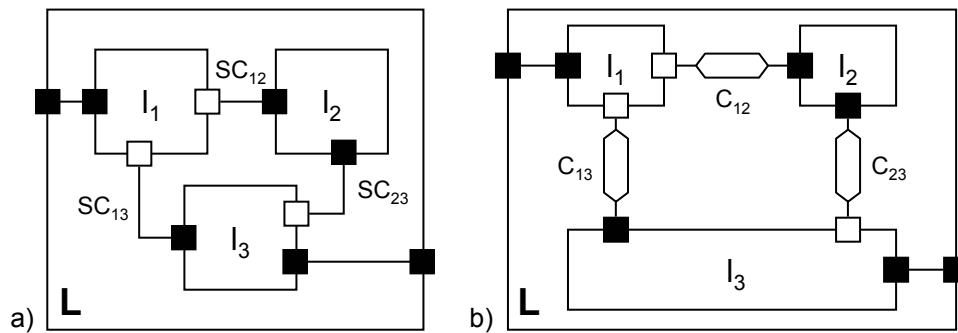


Figure 4.7: Equation 4.1 and 4.2 visualized: (a) functional breakup, (b) distribution

If the actor references I_1 , I_2 and I_3 are the same in figure 4.7 a) and b) (as demanded by the general identity requirement) then their actor classes must have been designed in such a way that they are capable to compensate the message loss modeled by the typed bindings. So, there are two aspects that shape the design of I_1 , I_2 and I_3 : (1) The actors represent functional fractions of the composite, actor L . That is their functional aspect. (2) At the same time, the actors have to handle the impacts on message transmission by the complex connectors. That is the communication aspect imposed by the fact of remote communication. We will turn to this two design aspects of a distributable component in chapter 5.

4.3 Networked Communication

For this very section, we would like to alert the reader. Although basically straight and simple, the author experienced people getting stuck with the following material. Our approach is that we look at any layer of a communication system independently. Like a surgeon we carefully cut an individual layer out of its environmental context of upper and lower layers and look at the individual layer as a network of its own right. This approach breaks with the traditional OSI way of thinking and arguing about systems design – and many people, especially experienced experts in the domain, seem to be “trapped” in this tradition. In OSI dominates the understanding that an upper layer is realized by services of a lower layer; some use the somewhat misleading terminology to say that the lower layer “implements” the upper layer. Following the OSI tradition, there is no notion to treat a layer as a completely independent unit.

The complex connector we introduced in the previous section is a first class abstraction; there is no helping construct needed such as a (lower) layer. OSI, on the opposite, defines the concept of a connection only in the context of layers: a connection is an association (cooperative relationship) established by the lower layer [ITU94]. Again, we are trapped in the OSI thinking of upper and lower layers.

4.3.1 Specialty of the Approach

In this section we view each layer in a communication system as a self-contained unit without any dependencies to other layers. Each layer unit consists of distributed entities communicating remotely to each other via a network that interconnects the entities, see figure 4.8.

From a methodological and engineering viewpoint we model each layer abstractly, modeling *what* the distributed layer entities do and modeling *what* kind of communication services and communication resources the entities use to bridge their spatial distance; for the time being we are not interested *how* this is achieved via a lower layer. That is, the model of a network is an abstract model of distribution, which includes a network topology (who is permitted to communicate with whom) and the used communication services (connectionless or connection-oriented).

Relation of Complex Connector and Communication Service

Where does the complex connector fit? What is the role of the communication service? The complex connector represents a modeling construct that abstractly emulates all effects of the impacts that remote communication between distant

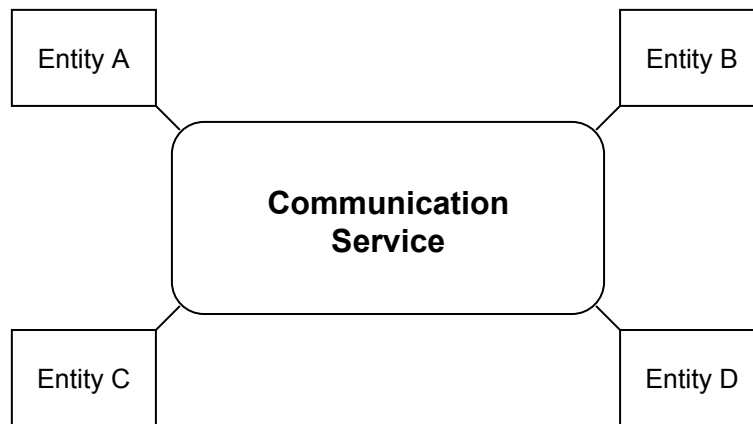


Figure 4.8: Modeling Distribution: Communication network of entities communicating via a communication service

parties may have on the communication. However, our realization of the complex connector, the typed binding, is limited in its use: it can be only statically tied to two ports of two actor references. How about establishing/releasing complex connections dynamically? We need something, which we can ask for inserting and removing a complex connector between any two ports at some point in time. The communication service fulfills this role. The communication service is an abstraction that realizes an omnipresent “meta-connector”: even though all communication parties are remote from each other, all of them have access to this “meta-connector” and can request a complex connector for communication. That is the power of abstraction! In our models we have access to a virtual service that is nothing we can explicitly point at in the real world!

To highlight this very special virtual service in our models, we use a slightly modified actor class symbol for it: we draw the actor class with rounded edges. The new symbol serves as a semantic mark of distinction.

Viewpoint on Subject of Study

In chapter 3 we studied TCP and UDP in a way the standard presents them (page 90 ff.). This chapter’s viewpoint suggests a different approach. Compared to figure 3.13 (page 90) we now model the communication among TCP and UDP users, respectively, as a self-contained communication network and the communication among TCP/UDP service providers as a self-contained communication network, see figure 4.9.

The glue that interconnects the user network and the provider network is subject of another main chapter. It is chapter 5 that introduces a precise definition of layering and shows different ways of smoothly interconnecting layers. With the

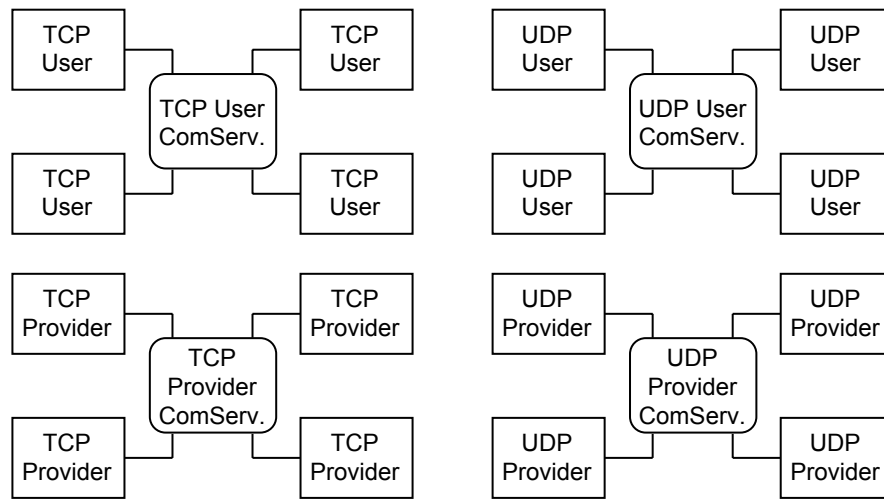


Figure 4.9: TCP and UDP from the viewpoint of networked communication of independent layers (compare to figure 3.13)

technique presented in the chapter about layering, we can perform the transition from figure 4.9 to figure 3.13 – as one option. Another sort of transition is also possible.

4.3.2 Connection-oriented Communication Networks

With the help of the complex connector we can describe *static* configurations of distant connection-oriented communication. The complex connector concentrates all impacts that the transmission may have on the messages to be conveyed. In reality, connections are rarely static; they are rather a form of a long-lasting dynamically created connection. Normally, connections are set-up and released on demand. The question is how we can model *dynamic* connections in ROOM that will be created and destroyed at run-time.

At first sight, the use of optional typed binding contracts seems to solve the problem. But that is just a halfway solution. Optional typed binding contracts are still static in the sense that the parties involved in the contract are fixed and given. On the other hand, we would like to stick to the approach shown in figure 4.8: The network should have all the meta knowledge about the network infrastructure and its topology, and should be in charge of connection establishment and release. And connections should be possible between any two parties as long as not in contradiction with the network topology.

The solution is to use the *import* feature for actor references. Imported actor references are a powerful tool that enables a modeler to described patterns of context an actor can be put in. An imported actor reference represents a certain

context an actor can be plugged-in during run-time; it is a context that co-exists with the context (the decomposition frame) the actor was incarnated in. A dynamic connection can then be interpreted as a context consisting of a typed binding contract and two imported actor references. Modeled that way, a dynamic connection describes a pattern of collaboration.

Modeling a Connection-Oriented Communication Network

Let us become concrete and take the TCP service user layer and model it as an independent connection-oriented communication network. Remind yourself that the TCP service user can be substantiated by HTTP [FGM⁺99], FTP [PR85], many other application protocols, by proprietary or standardized applications.

Figure 4.10 shows the TCP user network with the TCP users as the communicating entities. Via its port reference *c* the TCPComService can have many TCPUser actors. Each TCPUser has two port references: a control-oriented port reference *c* to request and release a connection service, and a data-oriented port reference *d* to send and receive data. So far, nothing has changed compared to the TCP model in chapter 3. New is that port reference *d* is unbound. Port *d* gets bound as soon as the TCPUser is imported as the initiator or acceptor of a connection request. In that case, TCPUser acts inside the TCPComService in an additional context; the context is given by the typed binding contract tcpConnector including the imported actor references. Note that the typed binding contract is fixed and not optional or anything else. Whenever a related initiator/acceptor pair is imported, the typed binding is incarnated and inserted to provide a channel for communication. Fixed typed bindings contracts are much more dynamic than the reader might have anticipated.

The whole point of modeling dynamic connections is a matter of delaying the binding of ports. This technique is independent of the fact whether the binding is of type ideal or not. The only way to realize this sort of late binding in ROOM is to describe the late binding as a collaboration of actor roles that define a plug-in context to interconnect unbound ports at run-time. What we just do to model the effects of remote communication is to exchange ideal (late) bindings via typed (late) bindings. In figure 4.10 the network can handle a maximum of *n* late bindings of type tcpConnector.

The coordination logic that accepts connection requests and releases via port instances of port reference *c* and synchronizes the interplay of control information between two ports is “hidden” in the behavior component of TCPComService. The behavior component also supervises whether a port incarnation of port reference *c*, say *c*[3], is permitted to coordinate with another port, say *c*[7]. This is the so-called network topology, which may impose restrictions on permitted pairs of user communication. We will come back to this in the next section about address-

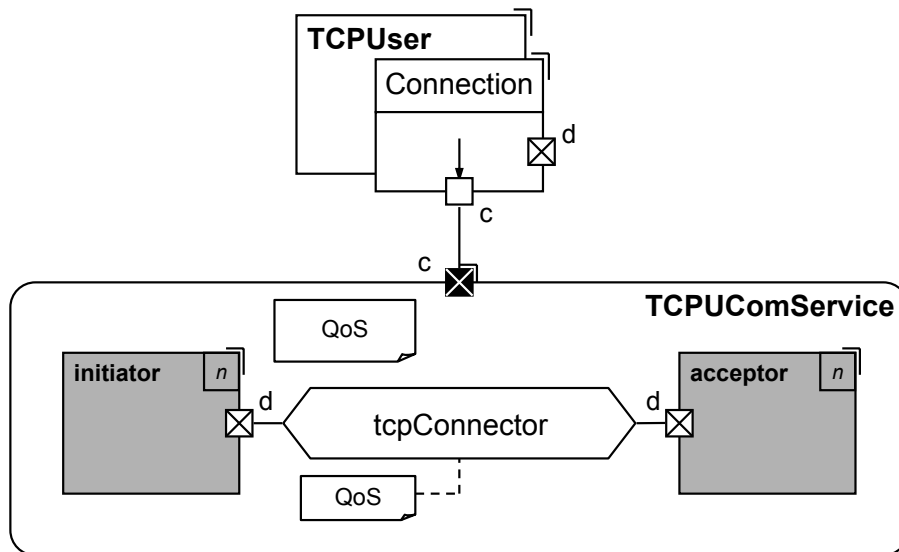


Figure 4.10: Model of the TCP user layer as an independent, abstract network

ing.

We could have added a port reference *m* on the boundary of **TCPComService** as a network management interface. Via *m* an administrator could, for example, dynamically change the topological configuration of the network, manually shut down connections to release hanging communication etc.

Note that for dynamic connections the network itself models QoS properties for the establishment and release phases of a connection. The typed binding models the QoS properties for message transmission, the network in addition may specify e.g. the time required to set-up or release a connection, the failure rate of successful connection requests, the tendency to spontaneously and unsolicitedly release a connection, which is a matter of connection reliability, and so on and so forth. It is the modellers responsibility, which properties are of interest for the purpose of the model.

As a service to the reader, the modeller may expose the general QoS attributes of the communication service and the specific QoS attributes of the complex connector via a note. The note attached to **tcpConnector** refers to the complex connector, the other note to the communication service **TCPComService**. We did not specify concrete values in the diagram in figure 4.10.

Remarks about **tcpConnector**

We have not said much about the details of a specification for **tcpConnector**. The reason is that the TCP complex connector represents an almost ideal message

transporter. Message loss is rarely enough to neglect it, the message order is preserved, so just the delay of message transfer may be of relevance. Per message the delay may vary from some milliseconds to several seconds. If network performance analysis is not the primary matter of concern for a system architecture model, it is usually sufficient to count in a constant time of delay or to neglect delays completely. Here, we assume a reliable complex connector of almost no delay. The complex TCP connector just converts incoming send messages to receive messages and vice versa.

Run-Time Behavior

A brief look at an incarnated version of figure 4.10 may help understand the presented specification from a run-time perspective. Note that the message schemata for port c and d remain the same as presented in chapter 3, see especially table 3.2 on page 93. So does the actor model of a TCP user, see figure 3.14 on page 94. A typical scenario of two TCP users establishing a TCP connection to communicate to each other is described subsequently and outlined in figure 4.11.

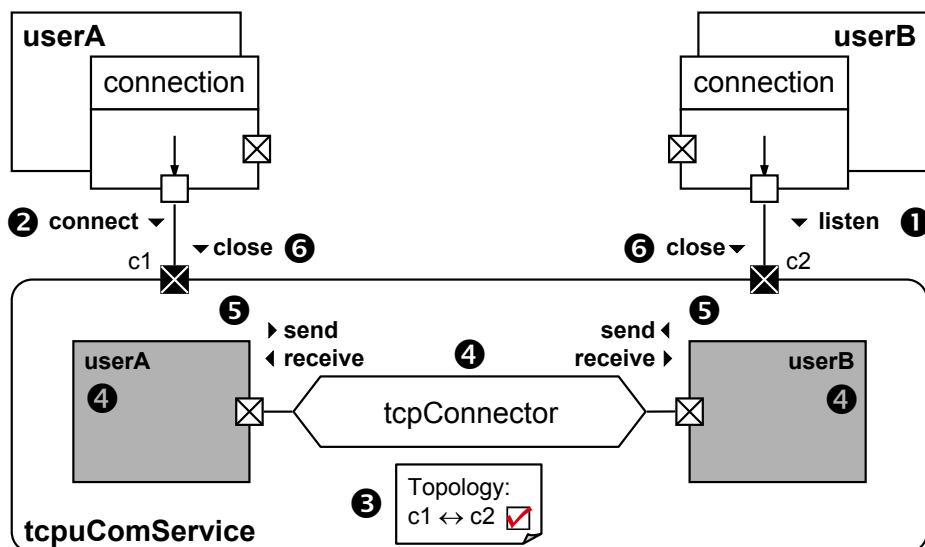


Figure 4.11: TCP user network, run-time scenario

- ❶ Actor userB requests a connection object via `getConnection()` from the behavior component and binds a local, free TCP port to the connection object. The TCP port (the local connection endpoint identifier, so to speak) is given as a parameter in `getConnection()`.² Then userB calls the `listen()` method and

²This initial procedure will be refined and become more clear in section 4.4.

waits for `listen()` to return. The return of `listen()` indicates that the connection is now ready for use. The return value is the address of the remote party's local connection endpoint, which includes the remote, local TCP port and the IP address. More about addressing issues can be found in a subsequent section.

- ② Actor `userA` also requests a connection object via `getConnection()` and binds a local, free TCP port to the connection object. Instead of listening for an connection invitation, `userA` actively tries to establish a connection via method `connect()` of the connection object. The address of the party `userA` wants to connect to, `userB` in this case, is handed over as a parameter to `connect()`. The address contains a remote local TCP port and an IP address.
- ③ The `tcpComService` actor receives the `listen` and `connect` messages with the corresponding parameters as message data. If a `connect` message refers to a TCP port/IP address pair for which the communication service actor has received a corresponding `listen` prior to that, then `tcpComService` sets up the scene to interconnect both users. Otherwise, `tcpComService` returns an unsuccessful `connectReply` message to `userA`. Of course, the prerequisite for interconnecting `userA` and `userB` is that the topology of the network permits communication.
- ④ If there are no objections against interconnecting `userA` and `userB`, `tcpComService` imports `userA` in the initiator context and `userB` in the acceptor context. Automatically, `ROOM` incarnates and establishes a typed binding `tcpConnector` between the initiator and the acceptor. For the import, the communication service must know the identifiers of `userA` and `userB`. In `ROOM`, the import primitive has the following format in C++:

```
result = aFrameSAP.import(actorId, actorRefName);
```

Thus, the actor identifiers must have been submitted along with the `listen` and the `connect` message. If both imports are successful, `tcpComService` sends a `listenReply` to `userB` and a `connectReply` to `userA`. As a result, the methods `listen()` and `connect()` return.

- ⑤ The import in `tcpComService` binds both ports of the respective network users. Now, `userA` and `userB` can use the methods `send()` and `receive()` to transmit and retrieve data. Method `send()` is translated into a `send` message. Method `receive()` blocks until an incoming message `receive` has been received.

- ⑥ If userA or userB closes or aborts the connection via `close()` or `abort()`, the connection object at the other side continues to exist (although not much usable anymore) until it is also explicitly closed or aborted. On receipt of a close or abort message, `tcpComService` immediately departs actor userA and userB from the connection context. The `tcpConnector` is not available anymore for data transmission.

Remark: Actually, there is a subtle difference between close and abort. When closed, outstanding data can be still received from the other actor, but nothing can be sent anymore. An abort abruptly and mercilessly releases the connection. Our model cannot capture this difference. A more elaborated model of TCP would unveil that a TCP connector is actually composed out of two uni-directional complex connectors. A `close()` only releases the connector in the sending direction (the connection is said to be “half-open” [Pos81b]), whereas an `abort()` quits the sending and the receiving connector. This refined model is a very good example illustrating that abstract models may put different emphasis on the aspects they model depending on the purpose of their use. If connection release handling is unimportant, our simplified model may be sufficient.

Some code that presents a possible implementation for userA and userB is shown in figure 2.5 and figure 2.6 (page 2.5 f.). Despite the fact that instead of a connection object a socket object is used in the two Python programs (sockets are a standard interface for TCP users), the scripts very well match the described behavior.

4.3.3 Connectionless Communication Networks

What is it that differs connection-oriented communication from connectionless communication? Basically, it is a matter of addressing. In connection-oriented communication, internal interfaces of communication are associated with a fixed (better: temporarily fixed) communication partner. Information that is pushed to the interface pours to the communication partner; and information the partner wants to notice us, pops up at the interface. In that sense, the interface is a sort of representation of the other party, and the interface identifier is an internal address denoting the other party. So, talking to another party requires to either use another interface (that is bound to the other party) or to newly bind the interface with the other communication party.

Figure 4.12 visualizes the addressing principle. The cloud symbols represent address spaces, here `space1` and `space2`. An address space is composed of individual addresses, `a1`, `a2` and so on. An address has to be unique with regards to its address space; an address is said to be *local* to the address space. An address can

be associated to another address of the same or another address space. The association must be unique and unidirectional. In figure 4.12, address a1 of space1 is associated to a2 of space2 and vice versa.

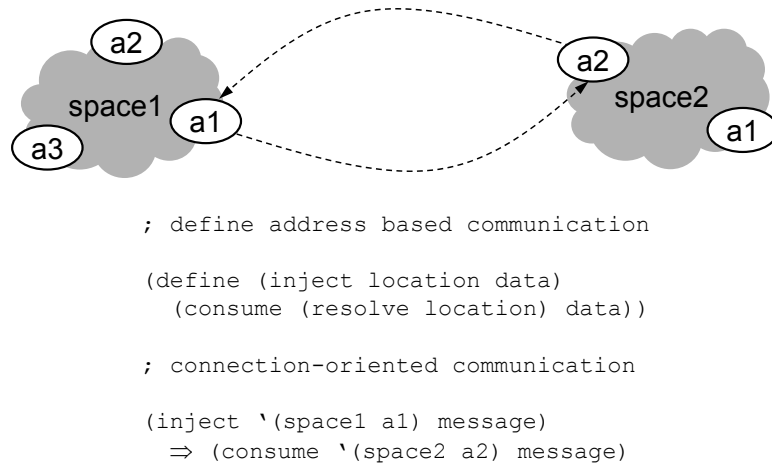


Figure 4.12: Address structure characterizing connection-oriented communication

Note that address associations do not represent connections, they just describe a structure of address relations. In a communication system, the address structure is used to direct the flow of information through the system. For the sake of demonstration, we define address based direction of information flow as follows: Data that is addressed to a particular address is actually addressed to its associated address. The definition in figure 4.12 puts it in the words of the language scheme [KCe98]: The procedure `inject` takes two arguments, location (a list of address space and address) and data, and is defined such that it is substituted by the procedure `consume`. Procedure `consume` takes also a location and a data parameter. The data parameter of `inject` is handed over as is, but the injected address is resolved to its associated counterpart. If an address/address space pair is not associated with another address/address space pair, it is by default associated to itself.

From an addressing perspective, the addressing structure underlying connection-orientation per definition demands that one address space is directly related to another address space via a pair of associated local addresses. That means, connection-oriented information flow can be expressed by a simple `inject` expression that evaluates to the `consume` expression of the associated address, see figure 4.12.

For connectionless communication the general addressing structure looks different. The arrangement of associations is so that two communication partners do not maintain direct relations between their address spaces, see figure 4.13. Instead, local addresses are associated to a third party, an external address space, `space3`.

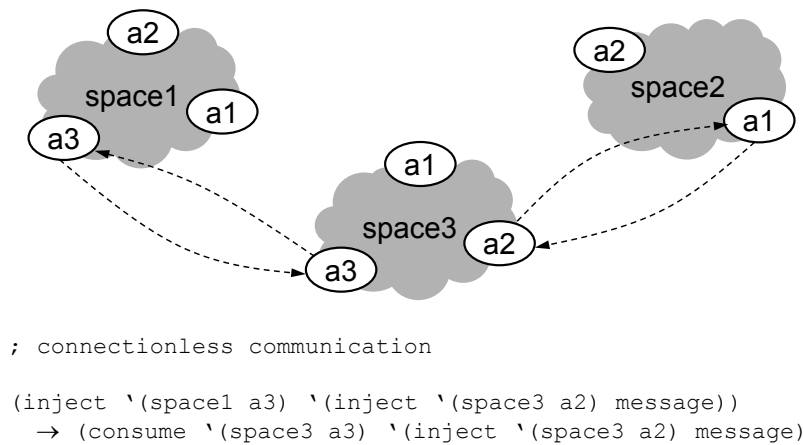


Figure 4.13: Address structure characterizing connectionless communication

To direct some flow of information from `a3` of `space1` to `a1` of `space2`, we have to inject a message that contains information how to further hand it over to its destination, see figure 4.13. In the example, the message is another inject expression that encapsulates the message to transfer. Method `consume` must be capable to distinguish “raw” data from data covered as instructions that demand message forwarding.

In reality, connectionless communication is rarely implemented by packaging instructions for the intermediate receiver; message schemata are used instead. Also the way to specify the destination address and procedures to resolve address associations look completely different. A very common procedure is *routing*, a technique used e.g. in the Internet to direct IP datagrams to their destination (see e.g. [Tan03]). However, our simple example illustrates the basic addressing principle that underlies connectionless communication: to transfer a chunk of information a local address and – in addition – an address outside the context of the local address space needs to be given. The reason is that connectionless communication is based on an address structure that uses a mediating address space. Furthermore, the mediating address space has to function in a certain way in order to forward messages. In the example we indicated that by an inject statement that needs to be understood by `space3`.

Consequently, users communicating connectionless need to have an internal representation of the address space outside their locally addressable scope. They need to specify the destination of their messages. Users communicating connection-oriented do not have to do that.

Modeling a Connectionless Communication Network

Again, let us become concrete and take the UDP service user layer and model it as an independent connectionless communication network. Much alike TCP users, the UDP service user can be substantiated by e.g. a DNS (Domain Name System) application [Moc87a, Moc87b], some other application protocols, by proprietary or standardized applications. The user layer network model for UDP is the connectionless counterpart of the TCP user layer network, which we modeled in the previous subsection. For comparison see also the upper diagrams in figure 4.9.

Actually, the model for a connectionless communication network is remarkable simple. The actor class representing a bunch of users, `UDPUser`, is connected to the `UDPComService`, see figure 4.14. That's it!

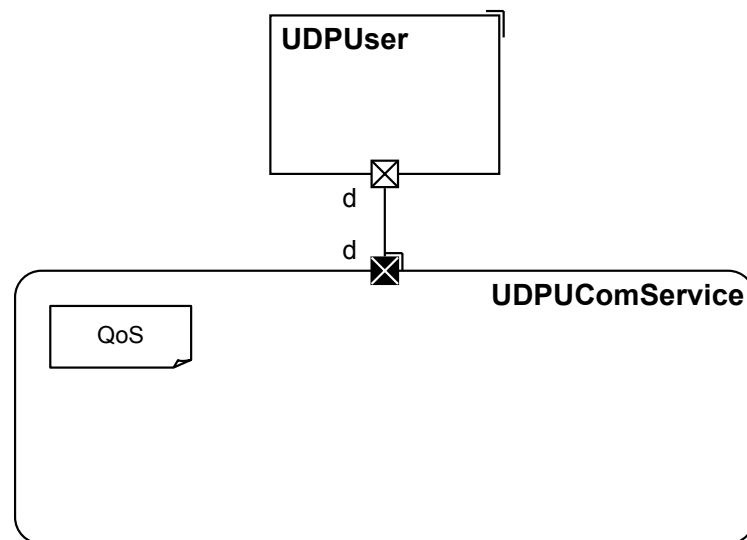


Figure 4.14: Model of the UDP user layer as an independent, abstract network

The addressing structure underlying connectionless communication as shown in figure 4.13 can be easily uncovered in the ROOM diagram. Identifiers of individual actor class instances map to address spaces, port identifiers of an actor map to addresses, and identifiers for individual bindings map to address associations. The `UDPComService` has the role of the external mediating address space with regards to the UDP users. More details about addressing can be found in section 4.4.

As an mediating unit, the `UDPComService` is responsible for fulfilling three functions: (1) take care of the network topology, (2) act as an distributor of data received from one UDP user to be sent to another UDP user, and (3) realizing QoS.

The network *topology* is as easy to model as in the connection-oriented case.

The UDPComService just needs to maintain a table that includes entries of permitted and non-permitted address pairs. Previously, a pair of addresses has been introduced by the notion of an address association. So, the network topology is described by a set of address associations that refer only to local addresses. For connectionless networks, the network topology describes potential paths of flow for *data-oriented* information. For dynamic connection-oriented networks the network topology describes the potential relations of interfaces that can be pairwise coordinated via *control-oriented* information. That is an important observation in preparation for the next section about addressing: Address associations *between* address spaces predefine paths of flow of information (be they data-oriented, protocol-oriented or control-oriented). On the contrary, address associations *inside* an address spaces (we will call this the *topology*) remain to be interpreted by the object being responsible for the address space. Internal address associations relate addresses and require further efforts to specify the meaning of that relation.

In its function as a *distributor*, the UDPComService takes the data received by a send message (see table 3.3 and table 3.4), checks the topology and sends with a receive message the data out at the port that was specified in the send message. The data is complemented by the address of the sender so that the receiver knows, who the sender was. The address information might be needed for further communication.

For that process the UDPComService models all *QoS properties* that describe the impact the transmission has on data. For UDP, QoS properties are absolutely relevant; the transmission is not ideal. Even though today's networks are extremely reliable in transmitting UDP data, the chance of message loss cannot be neglected. Also the time of data transmission may vary from case to case; a data package may appear as "overtaken" by another data package.

All three functions (topology, distribution, QoS) are implemented by the behavior component of UDPComService, they are partially made explicit in the model via notes. A proposal, how to annotate the topology is given in section 4.4.

Remarks about the "Empty" User Network

Some readers may wonder about the "empty" user network in the network model for connectionless communication. What about the functions we just mentioned, do they not require some internal data structures to store received data including addresses and possible some more attributes? Is it not a bit too little of information that we see when looking at diagrams like figure 4.14? How does one know that UDPComService does data transmission and not something else?

The reader is right – at least to some extent. If we are interested in an executable model, we need to specify ROOM models to a level of detail that contain

all these aspects. There is nothing to object against a refinement adapted to the level of abstraction the model should simulate. We even believe that refining an architecture to a rudimentary level of executability is to the benefit of the architecture understanding. The system architect starts to see the consequences of the architecture design on lower design levels. And, most important, the system architect is demanded to formulate *what* this specific architecture is supposed to do. Box diagrams alone leave much room for interpretation. Executability clarifies a lot.

Though we promote executable architectural models, we do not want to show all the details in the first shot. One can have lengthy discussions about “what architecture is”, but most people will agree to the statement that architecture provides a course grain view on a system. An architecture model should suppress details that are not of relevance for the understanding of the overall. If we argue from this point of view, figure 4.14 is at an appropriate level of suppressing details. On that level it is less interesting to see how the connectionless communication service is supposed to work; we just assume that it somehow does its job fairly well.

However, we have to admit that ROOM has some weaknesses to express architectural relevant information. For example, we cannot tag a ROOM actor class with a sort of stereotype (much like in the UML [HvW99]) and thereby declare its generic functionality. The only way in ROOM to mark an actor class specification as, say, a communication service (be it connection-oriented or connectionless) or a user component is by naming conventions. Otherwise we cannot be sure that the actor class is not designed to do something else. But naming conventions are a very bad semantic means to highlight architecturally relevant information in a system. Tagging would be helpful for classifying actor classes beyond using inheritance for the same purpose. We express tagging by slightly different notational symbols for actor classes. As already introduced, communication services are symbolized with rounded edges.

Another weakness of ROOM is its lack to annotate QoS attributes to model components. Most system designers would agree to regard QoS attributes of architecture relevance, so they should be visible in the diagrams. The way we handle QoS properties is by implementing them in the executable architecture model, but that is an inferior solution as discussed. That is why we sometimes highlight important QoS properties via annotations in the model as a helping construct.

Run-Time Behavior

A typical scenario of two UDP users sending data connectionless from one party to the other party is described subsequently and outlined in figure 4.15. Figure 4.15 is an incarnation of the ROOM diagram (figure 4.14) with two users, namely actor userA and actor userB. Actor udpuComService is an incarnation of the UDP user

communication service.

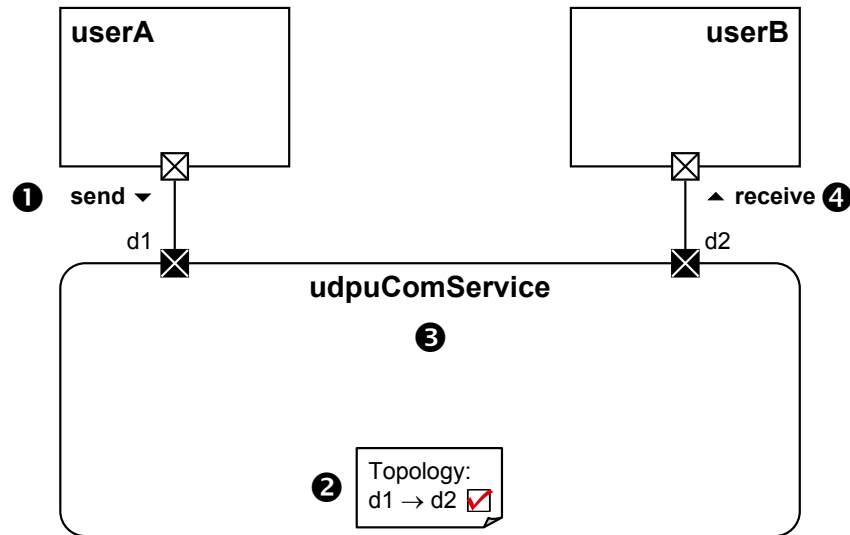


Figure 4.15: UDP user network, run-time scenario

- ❶ Actor userA sends a send message to udpuComService. The port identifier at which the communication service receives send is the *source address* of the message; in this case it is port d1. The communication service extracts the data and the given *destination address* delivered with send. The destination address must be the identifier of another port instance d.
- ❷ First, the udpuComService checks if the network topology permits data transfer from d1 to d2. In this scenario, the topology does not constrain the exchange of data.
- ❸ Then, the udpuComService has to apply a QoS model valid for the data package received by send. It may require to store a time stamp with the arrival of send in order to properly simulate transmission delay by delivery latency. Data may be completely erased from the udpuComService memory according to e.g. a statistical model of message loss.
- ❹ The communication service udpuComService delivers the data (if it has not been “lost”) to the receiver at port d2 via a receive Message.

More Models: The UDP/TCP Service Provider Layer

In chapter 3 we made first contact with TCP and UDP. The models in chapter 3 approached TCP and UDP as they are presented in the standards. We left open

what the dashed line on the provisioning layer stands for. In figure 3.18 the dashed line indicated the transport of TCP messages between TCP service providers, in figure 3.20 the dashed line indicated the transport of UDP messages between UDP service providers.

If we take this chapter's approach we can model the service provisioning layer independently of the upper user layer and independently of lower layers. Therefore, we re-interpret the UDP/TCP service providers as *users* of a provider communication service, see also the lower diagrams in figure 4.9. The provider communication service component models all the aspects of distribution. We basically only have to clarify the paradigm of communication (connection-oriented or connectionless), and the quality of services.

Both, the TCP standard and the UDP standard assume that the service provisioning layer uses connectionless means of exchanging data between individual service providers; in fact, the standards suggest to use IP, the Internet Protocol [Pos81a], as the lower layer protocol. Since we are not interested in the lower layer protocol but in an abstraction of its communication capabilities, we model the communication service interconnecting the service providers with the characteristics of IP. We equip the communication service actor class with IP-like interfaces and model the QoS accordingly.

Figure 4.16 a) depicts a model of the TCP service provider communication network and figure 4.16 b) depicts a model of the UDP service provider communication network.

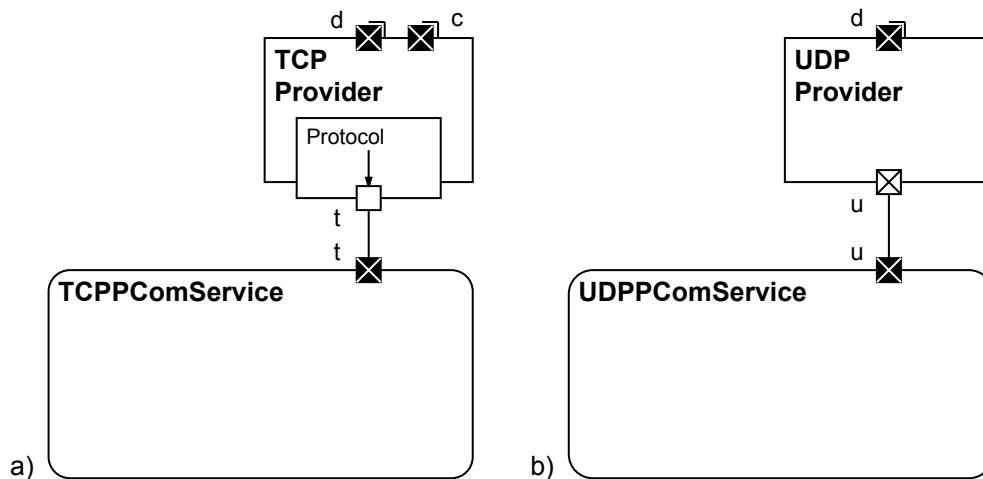


Figure 4.16: Model of the (a) TCP and (b) UDP provider layer as an independent, abstract network

The next main chapter discusses the relation of independently modeled communication networks.

4.3.4 Space-Based Communication Networks

There do exist other communication paradigms that are completely different in their general approach and in their way how they provide means of communication. Neither OSI nor the TCP/IP reference model are suitable frameworks for describing their overall organization. One of these different communication paradigms is *generative communication*. Generative communication has been introduced in the Linda programming system by GELERENTER almost 20 years ago [Gel95]. The basic idea is that a collection of independent (possibly distributed) processes make use of a globally shared and persistent dataspace of tuples for coordination and communication. A tuple is a tagged data record consisting of a number of typed field. Processes can put tuples into the shared dataspace (i.e. they generate communication records); but they can also extract or read tuples from the dataspace. Extraction and reading is in return of a search request. A search request specifies a search pattern by some of the values of the tuple fields a process is interested in.

In this section we demonstrate that our approach goes beyond the OSI RM and the TCP/IP RM and that the abstraction of a communication service can be perfectly adapted for space-based communication networks. We base our consideration on JavaSpaces, a “modern” implementation of GELERENTER’S tuple dataspace. For more information on the subject of distributed coordination-based systems, as TANENBAUM and VAN STEEN call them, please consult [TvS02, p.699ff.].

A Brief Summary on JavaSpaces

We will not give a full introduction to JavaSpaces but briefly summarize its service primitives. The interested reader may have a look at the specification documentation [SUN02].

In JavaSpaces, *entries* correspond to tuples. An entry is a typed group of objects. The following service primitives operate on the (entry) space:

write A *write* operation copies an entry to the space that can be used in future lookup operations.

read A *read* operation comes along with a look up *template*. A template is an entry that specifies *values* for some or all fields that need to be matched exactly and *wildcards* for remaining fields. The read operation either returns a copy of an entry in the space that matches the template or it returns an indication that no match was found.

take A *take* operation behaves like a read operation except that the entry is removed from the space.

notify On request, the space can *notify* the user when an entry matching a given template is written to the space.

The operations *read* and *take* appear in two variants: non-blocking and blocking. For the sake of brevity, we drop the *snapshot* primitive.

Modeling JavaSpaces

Modeling JavaSpaces on the abstract level just described is almost trivial, see figure 4.17; implementing JavaSpaces is a real challenge and faces a designer with almost all aspects of distributed systems design: data must be replicated and cached, distributed memories need to be synchronized, the response time might possibly have to satisfy real-time constraints, data must be reliably transported within the space etc. In so far, JavaSpaces is a good example demonstrating the gap between architecture and design: the architecture is (at least on a coarse granular level) easy to understand; it takes just a few minutes to learn JavaSpaces on that level and to use it. The implementation of JavaSpaces is at an expert level of skills and experience.

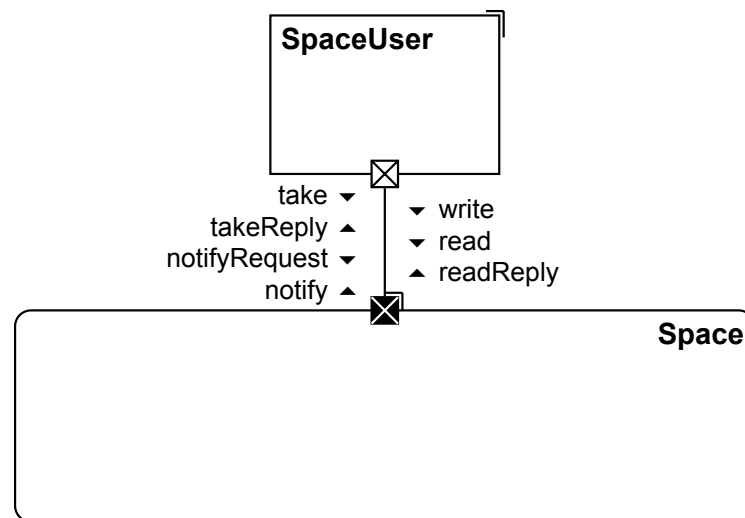


Figure 4.17: JavaSpaces user network model

In figure 4.17, the space is a communication service that processes entries (or tuples, if you like). The space actor is exactly that sort of an omnipresent, virtual communication service as which we introduced the concept of a communication service. A closer look at the specification of the behavior component would unveil its basic functioning. Here, we trust the written explanation given above. The Space is used by zero or more User actors. The message schema is annotated next to the binding. Astonishingly, it is that simple.

4.4 Addressing

Addressing is crucial to networking and a delicate issue for modeling. Generally speaking, addressing denotes a concept to identify and locate objects in a defined scope. So far, we more or less managed to bypass this issue. In the previous section we were confronted with some first thoughts about addressing in order to explain the differences of communication services. In this section, we study addressing systematically and in more detail.

4.4.1 Introducing Addressing Concepts in ROOM

Let us recap what we said about addressing so far: An *address* denotes a concept to identify and locate objects in a defined scope. The scope is the so-called *address space*, which is an assembly of addresses with each address being unique in the assembly. An *address association* relates two addresses to each other; the association is directed pointing from one address (the source address) to another address (the destination address). Source and destination address can but must not belong to the same address spaces. In so far, we can make a difference between *external* address associations and *internal* address associations. External address associations relate addresses of different address spaces, internal address associations relate addresses of the same address space.

Mapping to ROOM

Since addressing is a run-time notion (observe, an address identifies and locates an *object*, which is a run-time entity), we have to think about how the conceptions of address, address space and address association apply to run-time conceptions in ROOM. Remember that actor, port, and binding are the run-time materializations of actor class, port (class), and binding (contract). So the question is what an address identifies: an actor, a port or a binding? What, in consequence, does an address space correspond to? What about address associations?

We define ports to be the objects that are identified and located by addresses. Since an actor establishes the context within which a number of uniquely different ports reside, an actor represents the scope of the addresses, the address space. In the way bindings relate ports, address associations relate addresses. So, bindings implement external address associations. To be precise, a binding maps to two address associations, one in each direction from one address space to the other address space. The protocol associated with the binding specifies whether both address associations are used or not. Internal address associations are realized by the behavior component of the actor. Table 4.1 briefly summarizes the mapping.

Given this mapping, figure 4.12 (page 136) and figure 4.13 (page 137) could be easily redrawn as ROOM run-time diagrams.

Table 4.1: Mapping addressing conceptions to ROOM run-time conceptions

Addressing Conception	ROOM Conception
address	port
address space	actor
address association (external)	binding
address association (internal)	inside behavior component

By default, ROOM has no built-in support to relate a port to an address and use that address to identify the port. Instead of thinking about a possible language adaption to ROOM, we follow the conservative strategy to change as little as possible in ROOM and leave it to the implementor of the ROOM actor to manually hard-code the address/port relation. Such a relation could be implemented by a table e.g. in form of an associative array, called directories in the Python programming language.

More of a problem is how we annotate address/port relations in a model *specification*. We are now back on the “class” level of a ROOM model and leave the run-time object level. Surely, address information is of architectural relevance in a distributed communication system – it should be made visible although we handle the port/address relation internally, inside the actor.

Notation

The proposal is to annotate address information outside the actor class symbol but in conjunction with the port class it refers to, see figure 4.18. Usually, addresses are of a certain type and format. For example, addresses could be of type “IP address”, which is a 32-bit number that is usually written in dotted decimal notation (like 137.226.168.58). Or, another example, addresses could be of type “TCP port” or “UDP port”, which is a 16-bit number. Address types are best modeled by data classes in ROOM. Thus, the address information attached to the port class consists of the name of the data class used to model the address type and precedes the port class name. The optional port class name and the data class name are separated by a colon; this is the typical type notation used in many other modeling and programming languages. In fact, what we do is introducing a type concept for ports, even though we do not strictly implement it as that. In addition, we may specify actual values, which become assigned to the address data object

at run-time. The list of values is given in curly braces and determines the maximum number of ports to be incarnated. For the sake of brevity we will often use symbolic names in our ROOM diagrams instead of “real” values.

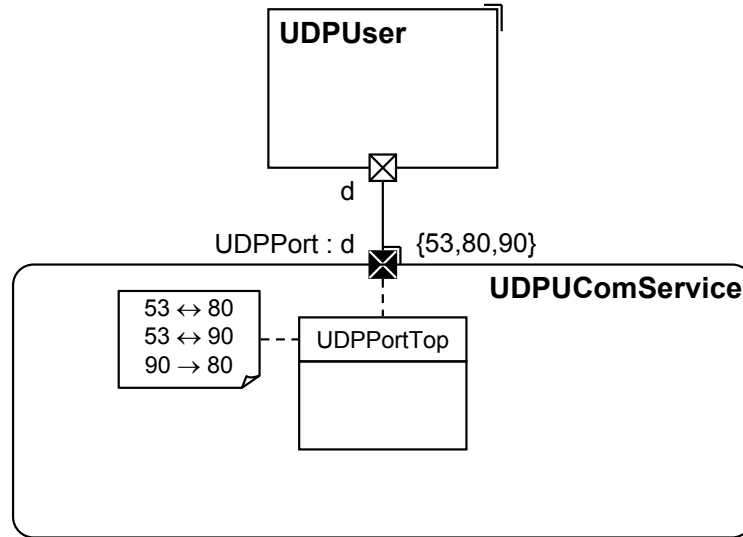


Figure 4.18: The notation for addresses and address associations exemplified

As was mentioned, external address associations are specified via binding contracts. So there is nothing to concern about. However, if we want to annotate internal address associations in order to model an *address topology*, we propose to use a data class for that. The topology data class is visualized inside the actor class symbol. The port class(es) it refers to is shown by a connecting dashed line. Of course, there can be more than one topology data class inside an actor class. Informally, notes next to the data class symbol may list permitted or non-permitted pairs of address associations, so that one gains a clear picture of the address topology without digging into the details of the data class implementation. For a connectionless communication service, the address topology describes the internal information flow of the data received. For connection-oriented communication, the topology describes the possibility to synchronize endpoints of control-oriented information.

Figure 4.18 exemplifies the notational proposal discussed so far. The connectionless UDPComService has a port class *d* that is related to address data class UDPPort. One could also say that port class *d* is of type UDPPort. Next to the port class we find a list of UDP ports that become assigned as concrete values for the port class if incarnated. Externally, the UDP port class is associated with the port class of a not further specified UDPUser actor class. Internally, the data class UDPPortTop specifies the topology for connectionless communication. In

the example, only permitted relations are listed; Port 80 cannot communicate to Port 90.

For one more time, we see that information that is internal to a ROOM actor and – according to the ROOM philosophy – invisible to the outside is made public. This grey-box style of publishing internal actor information becomes a “trademark” for meaningful ROOM models on an architecture level. Architecture modeling is not only about structure and overall organization of a system in modules or components; it is also not only about precisely specifying interfaces. To some extent, architecture models live in the “grey” border area exposing internal design information as architecturally relevant information and vice versa.

Handling Asymmetry

Another issue we have to solve is the asymmetry of addressing. Look at the previous figure, figure 4.18. It is the communication service specifying the address space of UDP ports not the UDP user. No other modeling option is available. The communication service has to model the abstraction of transporting data in a distributed system; this includes the knowledge of the communication topology and the ability to identify and locate, namely to address users. However, UDPU-ComService cannot look beyond its “addressed” ports, it does not know, who is actually connected to, say, UDP port 53; it does not know if the right user is connected to that port. On the other side, the users would like to be related to certain UDP ports. For instance, a DNS (Domain Name System) [Moc87a, Moc87b] application demands UDP port 53 because it is the standardized UDP port number other applications expect to find it attached to.

Getting the addressing asymmetry of the model in “balance” is primarily a run-time issue. It requires to extend the message schema on the *d* ports by registration messages. One of the first actions the users of the communication service have to do is to send out a message and ask, for example, “Could you address me under UDP port number 53, please?”. If the port number is available, the communication service reserves number 53 as a concrete addressing value for the ROOM port the request was received at. The communication service’s response “Yes, I reserved UDP port number 53 for you” indicates successful registration to the user. After the registration procedure, other users can now rely on that information addressed to UDP port 53 will be *directed* towards the “right” user. If it will *reach* its destination is a matter of the QoS properties specified for the communication service.

For the run-time scenarios we described from page 133 on for a TCP user network and from page 140 on for a UDP user network to be complete, we have to insert a step preceding ❶:

- Actor userA and userB register themselves by calling the bind() Method. As a parameter to bind() a TCP/UDP port number is specified; it is the port number the user would like to be addressed by. The return value of bind() indicates the outcome of the ROOM port to TCP/UDP port binding. If it has been successful, the communication service accepts other requests from the user.

The bind() method is actually a method used on the user to communication service interface, see e.g. the Python script on page 40. To make this work in our models for TCP and UDP we have to extend the list of service primitives and service messages by one more primitive and two more messages, see table 4.2. This table is to be regarded as an addition to table 3.1 on page 92 for TCP, and to table 3.4 on page 102 for UDP.

Table 4.2: Binding addresses to ports: service primitives and service messages

Primitive	Message
BIND	bind bindReply

The bind extension can be regarded as a configuration procedure. An architecture model that should be executable must implement at least a rudimentary configuration procedure. An additional or alternative possibility is to introduce a management interface for users as well as for the communication service and configure the system through these.

Addressing: Architecture vs. Design

It is a valid question to ask, why we do not relate port d of the UDPUser Actor with an UDP port data class as well, see figure 4.18? This is a very hairy question, for which it is hard to give a definitive answer, because it touches the area of “What is architecture, what is design?”. The borderline between architecture and design is hard to draw and academia still has not helped getting that clear, some even wonder if it at all exists. We would like to give the following answer: In the ROOM diagrams we use for modeling the system architecture of communication systems, we aim to suppress details that are not relevant on an architecture level and try to resist the temptation to show fine-grained details that are nice to know but not essential on the architecture level. Addressing is of architecture relevance, no doubt about that, and addresses have to be related to ports of the communication service for that very reason. As a result, architecture models look asymmetrically

tagged with addresses (as we named it), which calls for balancing. The balancing is an implication of the architecture model and an implication is something we do not need to further highlight. We regard the realization of such implications a design issue. Otherwise, we could regard our model as sort of overspecified and, even worse, the architectural relevant information gets blurred: if the user port class is related to an address data class as well, we cannot clearly distinguish anymore who addresses whom from an architecture point of view.

This line of argumentation is thoroughly followed throughout the whole work. For example, in chapter 3, we argued similarly to explicate the Controlled Domain Model (CDM) only on the user side and not on the provider side. The user's CDM implies a similar or reused domain model on the provider side, but that is not in need to be shown. This "asymmetry" is a characteristic of our architecture models. We take the view that the implied consequences of that asymmetry are a matter of the design level and get resolved there to fully balanced "symmetrical" design models. Though we have to admit that striving for executable architecture models puts the modeller in a permanent risk to pass the border from asymmetry to symmetry. Getting an architecture model to execute requires to design behavior, and that often means to spell out the hidden half of an "asymmetrical" model.

4.4.2 Modeling Address Hierarchies

Until now, we did not consider the full addressing scheme, TCP and UDP users use to address their communication partner and for being addressed by their communication partner. For the sake of simplicity, we restricted ourselves to TCP and UDP ports, respectively. In reality, the full addresses consist not only of a port number but also of an IP address. IP address and port number cascade address spaces in very much the same way as the postal system cascades the address spaces "postal code" and "street number" with "TCP" or "UDP" as the street name.³ The resulting hierarchies of address spaces are a common means to organize and structure the whole space of "locations" in a communication system.

There are two options to model address hierarchies: (1) Introduce a new address data class that is composed of an IP address data class and a port data class; use the new data class as a type for the port class of the communication service. (2) Model the hierarchy of address spaces via actor classes.

Both options are equally valid approaches. The first option is shorter, the second option more expressive. Since the first option (introduce a new address data class) is nothing new and has been discussed quite extensively, we will show what the second option (model the address space hierarchy) brings to us.

³The analogy of a street name corresponds to what is called a *protocol identifier*. We will comment on this later.

Revisiting the TCP User Network Model

Figure 4.19 is an more elaborated model of the TCP user network; it is not very much different from the previous version, see figure 4.10 on page 132. The TCPUComService gets a hold on the IP address space. Port c of TCPUComService is now bound to IP addresses and is connected – via a binding – to an upper actor class TCPUDeMux. This actor class models the TCP port address space, and addresses via its c port an TCP user.

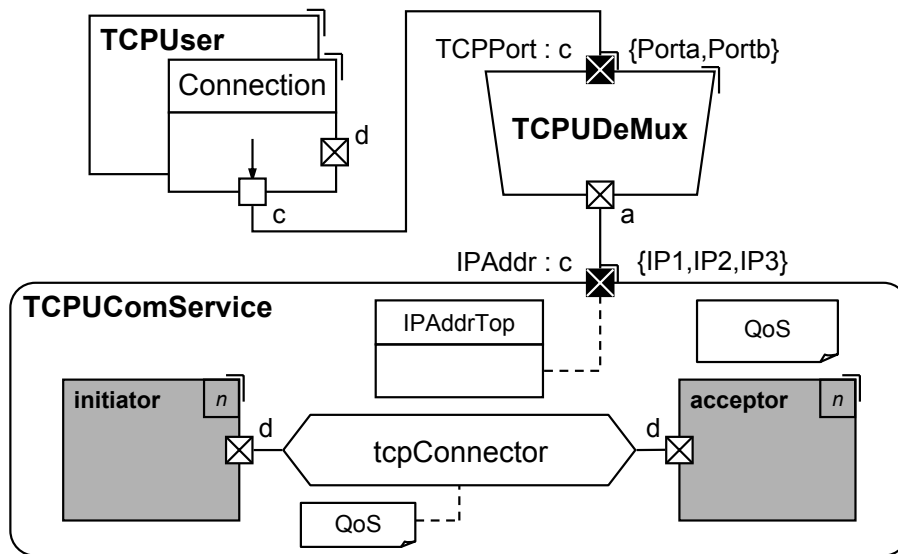


Figure 4.19: Elaborated model for TCP user network

This separation of address spaces also results in a separation of internal address topologies. The topology of the TCP port address space is completely independent of the IP address topology. This is a quite good logical abstraction of how TCP user networks work in the “real” world. The function of the TCPUDeMux actor is that of a multiplexer/demultiplexer:⁴ to serialize incoming messages from the different users down to the communication service, and to spread the incoming message stream from the communication service to the right port. Since our models are designed to be executable, a look into the implementation of TCPUDeMux would give us a more precise description of its functionality. Plain text, like our description, is always just a helping communication means among human beings.

The de-/multiplexer (deMux) is such a standard component in a telecommunication system that we want to declare actor classes of that kind. In our models we use a special notation for de-/multiplexers: the symbol of a deMux actor class

⁴The name “demux” is a common abbreviation in telecommunications; it combines and shortens the word “multiplexer” and “demultiplexer”.

shapes like a trapezoid. If possible, the port that has the multiplexed stream as outgoing is put on the short edge of the trapezoid and the port that has the demultiplexed stream as outgoing is put on the long edge.

When incarnated, the ROOM model shown in figure 4.19 “unfolds” to a tree structure. An incarnation of the TCPComService is at the root that forks up to an TCPDeMux actor per IP address, which in turn forks up to a TCPUser actor as a leaf per TCP port. Each actor in this tree is running independently in its own thread of control. ROOM makes it very easy to specify concurrently executing components.

If we want, we can add an architecture detail to the TCPDeMux actor. In figure 4.20, the TCPDeMux actor contains the same connection pattern as the TCPComService actor does with a minor but important difference: the typed binding has been exchanged by an ideal binding. What this refined model says, is that data sent from one TCP user to another TCP user within the same TCP port address space (i.e. the two users have the same IP address) is conveyed without any impacts of a non-ideal transmission medium. This refined model also matches quite good with reality. If you have two TCP user applications running on your Personal Computer (PC) that send data towards each other, the data messages do not leave your PC but take a “short-cut circuit” inside. So one should not get fooled by the simplicity of the basic function that is associated with the symbol for a deMux or any other modeling entity. The hard fact is: the functionality can be quite complex (see figure 4.20) even though the basic functioning is easily eye-catched (see figure 4.19).

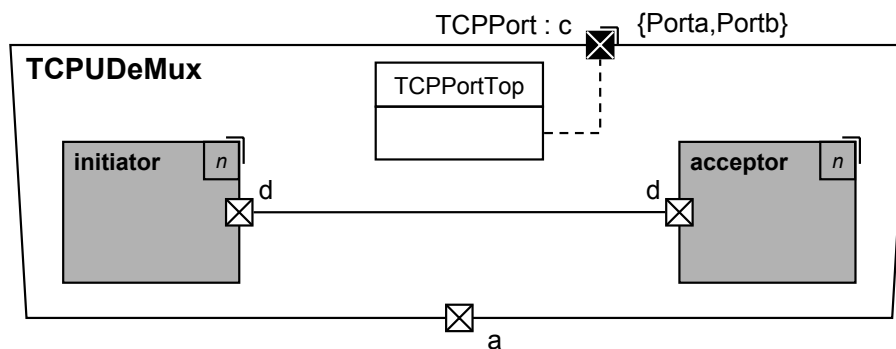


Figure 4.20: Possible refinement for TCPDeMux

Revisiting the UDP User Network Model

Having come that far, it is almost trivial to model the UDP user network. Figure 4.21 shows an elaborated version of figure 4.14 (page 138) without further commentary.

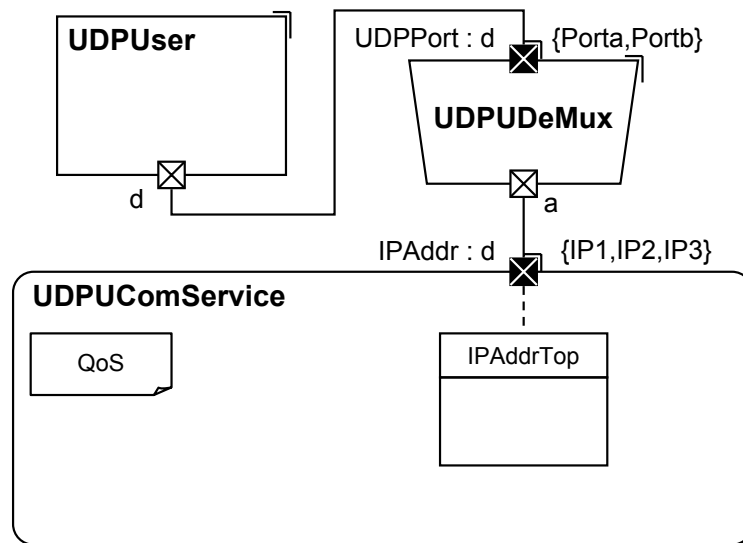


Figure 4.21: Elaborated model for UDP user network

Where has the Protocol Identifier been?

Readers familiar with IP, TCP and UDP may wonder, why we silently dropped the Protocol Identifier (PI) in our address space hierarchy. Where has the protocol identifier of IP been? The answer is already in the question: The protocol identifier is a discrimination mechanism of the Internet Protocol, and we are not modeling IP here! This was the basic assumption of this chapter: we model a layer as an independent network. Since the models we looked at in this chapter so far abstractly describe TCP and UDP user communication networks we cannot expect to be confronted with something that relates to IP.

As a matter of fact, the PI is implicitly existent in our models, but most likely not in the way most experts would expect it according to their traditional education in communication systems and computer networks. We are having two models, a UDP and a TCP user network model, and these models are the two streets – if we refer to our previous postal service analogy! So, if you have the street you just need to know in which city (IP address) and at which street number (TCP/UDP port number) you have to knock on the door. It may seem strange at first sight, but its true and consistent: the TCP user network model and the UDP user network model are representatives of two different protocol identifiers.

Revisiting the TCP/UDP Provider Network Model

If we move to another layer, the TCP/UDP provider network, and organize the two models of figure 4.16 a bit differently using an intermediate address space,

the notion of a protocol identifier almost naturally pops up.

When we modeled the TCP and the UDP provider network (page 142), we mentioned that both providers make use of the same connectionless communication service. Let us assume (as we did) that in both cases the addressing is done via IP addresses. Then, why not aim for reuse and connect both providers to the same communication service?! If we do so we need some means to discriminate the TCP provider and the UDP provider. We can achieve this with an additional address space that functionally acts like a de-/multiplexer, see figure 4.22.

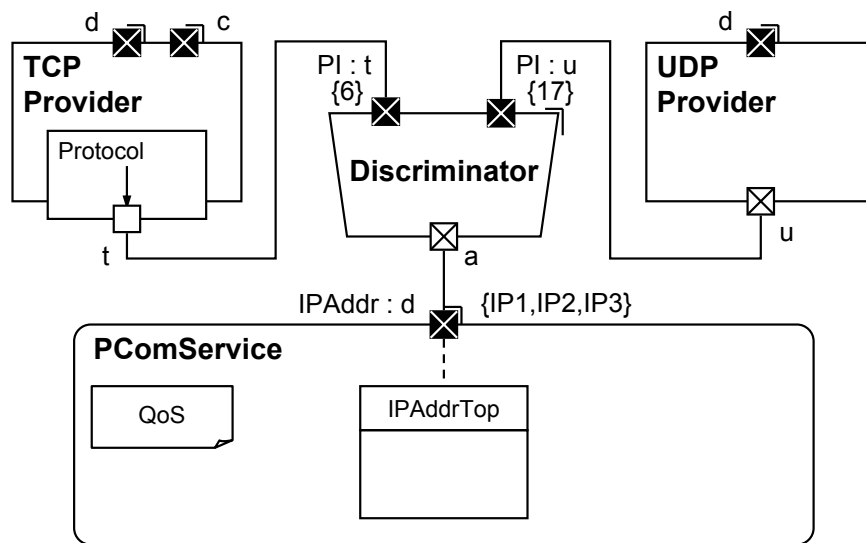


Figure 4.22: Combined model for the TCP/UDP provider network

The Discriminator actor in the figure has on its upper side two port classes *t* and *u*, which are related to a PI (Protocol Identifier) data class. Actually, the protocol identifier is a standardized number, which is 6 for TCP and 17 for UDP.⁵

⁵See <http://www.iana.org>

4.5 Summary

This chapter’s main theme, the aspect of distribution in a communication system, was first approached from a theoretical viewpoint: we mathematically defined a construct, called *complex connector*, as an element that (a) represents a line of communication between distributed entities, and (b) abstracts away all effects remote communication can have on the quality of communication by a set of properties called *Quality of Service* (QoS). Our definition of distribution, condensed by formula 4.1 and formula 4.2, is much more general but also more precise than most definitions to be found in literature.

We then introduced the complex connector in ROOM. Therefore, we extended ROOM’s concept of a binding. A *typed binding* is much like a binding and refers, in addition, to an actor class that specifies the “interna” of the typed binding (like e.g. QoS properties). The notation for a typed binding is shown in figure 4.23. The binding name is optional.

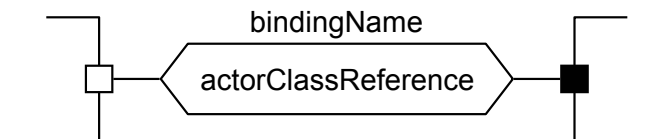


Figure 4.23: Notation for typed binding

The (typed) binding is a contractual conception that ties two ports of two actors together. Problematic is that the contract is static. To insert and delete typed bindings dynamically, the notion of a communication service comes in handy. A communication service is a sort of omnipresent “meta-service” all users are “magically” connected to despite their distribution. Users can request and release complex connectors from the communication service. In case the communication is connectionless, the communication service deals with forwarding individual messages.

The communication service is key in modeling each layer in a communication system as a self-contained network without dependencies to other layers. The communication service is a, so to speak, stand alone abstraction. Because of its special semantic role, the actor class realizing the communication service is drawn with rounded edges.

In this chapter we exemplified communication networks on TCP and UDP. We modeled each layer, the TCP/UDP user layer and the TCP/UDP provider layer, independently. If we generalize the examples, we see that the focus is on the communication service and less on the user connected to the service (be it a TCP/UDP user or provider or any other sort of user). That is why we can state that a network of distributed entities communicates to each other either via a connection-oriented

(CON) communication service or a connectionless (CNL) communication service. Both cases are diagrammatically depicted in figure 4.24. Other communication paradigms, like e.g. space-based communication, can be also modeled according to our approach, but they are rather the exemption than the general case.

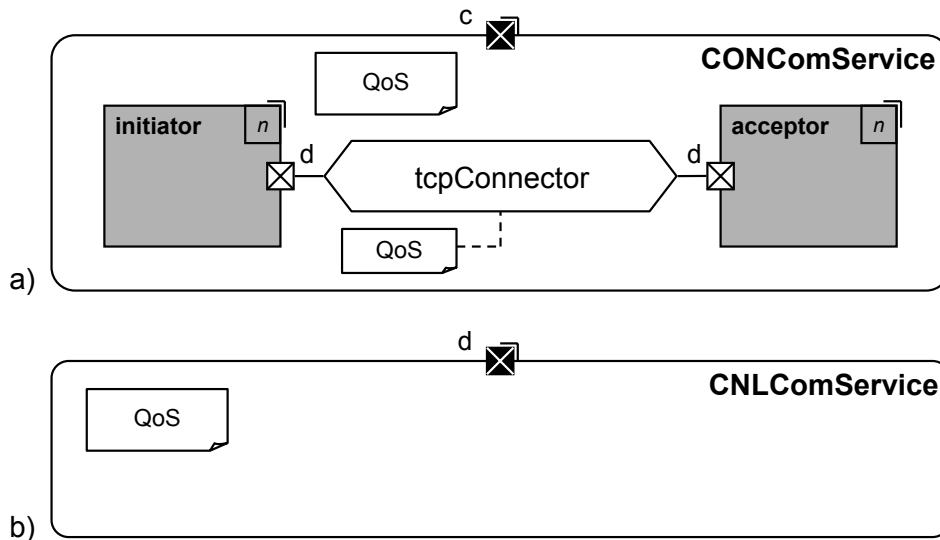


Figure 4.24: Generalization of communication services: (a) connection-oriented (CON), (b) connectionless (CNL)

Figure 4.24 a) shows a connection-oriented communication service. Users of the service get imported either as initiators or acceptors of a connection request at the corresponding placeholders. Via this collaboration pattern, two users can be bound late into a communication context. The typed binding models all QoS of the communication path between initiator and acceptor. Users of `CONComService` do not only need to have a port that connects them to port `c` but also a port for sending and receiving data, a port `d` for the connection context. Figure 4.24 b) shows a connectionless communication service. In connectionless communication, senders of data have to provide the receiver's address along with the data. The communication service `CNLComService` models QoS (like transmission latency, reliability, etc.) and delivers the data to the right port associated with the address.

We also learned in this chapter how to model addresses, address spaces and address associations. To model addresses, we introduced a typing concept to ports. A port can be associated with a data class that captures the address type. Address spaces are almost naturally determined by the actor class concept. (Typed) bindings realize external address associations, data classes modeling an actors address topology realize internal address associations. As an example, see figure 4.25, a not further specified communication service gets hold of the IP address space.

IPAddr is the type of port *c*, IP1, IP2 and IP3 are a list of three possible address incarnations of port *c*. The internal topology of the address space is modeled by data class IPAddrTop. A dashed line indicates to which port the topology refers to. The DeMux actor models another address space “on top” of the IP address space; in this case it represents an address space of ports. Since the de-/multiplexer is such a standard component in telecommunication systems, we introduced the shape of a trapezoid as a special notational symbol for it.

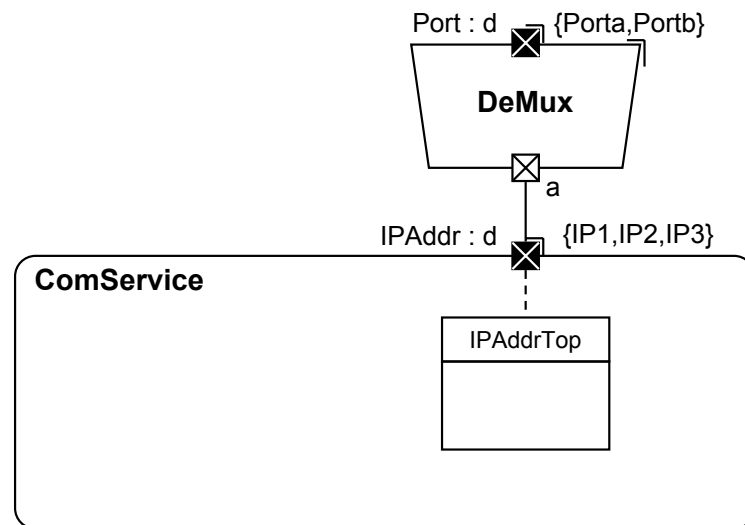


Figure 4.25: Example for modeling addresses, address spaces and address association

In this chapter, we solely talked about distribution. We were not really in need of the notion of a layer, we even did not talk about horizontal or vertical communication. Somehow, it looks like that this chapter has no link to the previous chapter about communication types. The glue, that puts everything together and relates independently modeled networks of communicating entities, is to be found in the next chapter about layering.

Chapter 5

Layering

In the previous chapter, we looked at each layer independently: A set of users communicates to each other via the abstraction of a omnipresent communication service. The question now is how several layers of communication networks are interconnected and make up a layered system.

In section 5.1, we provide an algebraic definition for the notion of layering in telecommunication systems. This definition is key to a simple but very powerful technique to relate different communication networks to build up a more complex architecture. In section 5.2, layering in communication systems is contrasted to the traditional understanding of layering, which is investigated on ROOM's layering concepts. As a result some improvements to ROOM are proposed and introduced. Some examples of layered communication networks are studied in section 5.3; the examples are based on the material from previous chapters. In section 5.4, we introduce the notion of planes to our architectural tool set. Section 5.5 closes this chapter with a summary.

5.1 What is Layering?

The most striking observation regarding layers in telecommunications is that the SAP is the key concept (see also figure 2.2 on page 32). OSI does not reveal how this concept can be notated or visualized, nor does it formally define the semantics of the SAP concept. Early examples of capturing the SAP concept can be found in the standard of the Specification and Description Language (SDL) [ITU99a]. Since SDL was developed with a strong impetus of the telecommunication industry, the designers of SDL provided some application suggestions for typical telecommunication problems. The SAP was no exception. The suggestion that is given as an example for layering by service access “modeling” is described in [ITU93g] and elaborated on in [EHS97]. It is distinguished between a service user and a service provider but the service access is reduced to signal lists to specify the SAP interface. The behavioral part of the SAP is hidden in the process of the service block owned by the service provider. No clear distinction is made between the internal interface logic of the SAP and the service logic. The example indicates that although the SAP has been a well-known concept for quite some time now (more than 15 years) it does not seem to be sufficiently understood as a modeling concept, which requires some more attention and careful design.

5.1.1 Definitions in Literature

Layering is one of the oldest techniques in software engineering to structure a system. Possibly the first, who made systematically use of layering was *Dijkstra*; he used layering for the design of the THE operating system [Dij68]. Even though “layering” is a standard term among system architects, the definitions and explanations given vary in their details and leave a confusing notion of what layering precisely is. A representative explanation for layering is the one from SHAW and GARLAN [SG96, p.25]:

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. [...] The most widely known examples of this kind of architectural style are layered communication protocols. [...]

SHAW and GARLAN highlight that layering is a means to organize a system. That seems to be the greatest common “divisor” in literature. Some authors interpret the service relationship as a “use” dependency between layers, see e.g. [HNS00]. Opinions vary about the kind of organization: Is it hierarchical (what does it mean?), is it abstracting (see e.g. [OMG01]), or something else (see e.g. [CBB⁺03])?

Interesting is that communication systems are regarded as a standard example of layered systems. As a matter of fact, reading e.g. TANENBAUM [Tan96, p.17] sounds familiar with the quote above in mind:

[...] most networks are organized as a series of layers or level, each one built upon the one below it. [...] the purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented.

We believe this comparison, layering in communication systems and layering as a structuring principle, to be wrong. Subsequently, we will provide a mathematical definition about what layering in a communication system is and contrast this to layering as an organizational means as it is used in ROOM. The essence is that “layering” in communication systems is a refinement relationship, whereas layering in the traditional, the organizational sense is a kind of “use” relationship.

5.1.2 An Algebraic Model of Layering

For mathematical treatment, we break our problem (What is layering?) into its constituting building blocks. For the sake of lesser complexity, we neglect network configuration and set-up and ignore the dynamics of establishing and releasing complex connectors. Instead, we assume a network to be static, so that a set of users communicate to each other directly via complex connectors. That means, our mathematical formalism can be linked up with the formal model of the preceding chapter about distribution. Furthermore, we restrict our consideration to only two communicating parties.

In the preceding chapter, we viewed each network of communicating entities independent of any other network of communicating entities. The notion of *layering* sets two communication networks in context and relates them in a specific way. Figure 5.1 schematically picks up the situation. There are two communication networks composed of two entities L' and L'' with an interconnecting complex connector in the middle. The upper communication network is said to constitute layer (N), the lower communication to constitute layer ($N - 1$); the layer denotation is used as an index.

Figure 5.1 shows how layering works according to OSI RM and the TCP/IP RM. The diagram serves as a basis for the subsequent discussion. We mainly use OSI terminology, see section 2.1. Compare figure 5.1 also to figure 2.2 (page 32); both are closely interrelated.

In the general case, the layer (N) peer entities send and receive PDUs (Protocol Data Units) over a complex connector C . So do layer ($N - 1$) peer entities over a complex connector \hat{C} . *If layered, the upper complex connector C is taken*

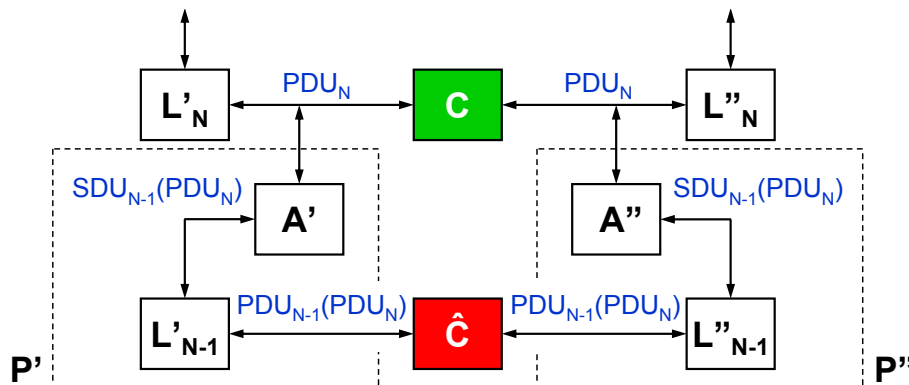


Figure 5.1: A schematic model of layering according to OSI RM and TCP/IP RM

out from the scenario. The PDUs sent and received from layer (N) entities are redirected via a layer adaptor A . The layer adaptors take care of the conversion of a PDU to one or more SDUs (Service Data Units) and vice versa. A PDU can be broken up into several SDUs in case of PDU fragmentation. In one direction, from top to bottom, the SDUs are fed to the lower layer entity, the lower layer entity takes the PDU, packages it into a PDU of the lower layer and sends it over the lower layer complex connector \hat{C} to the other side. In the opposite direction, bottom up, the upper layer PDU is retrieved from the lower layer PDU and handed over to the upper layer entity inside an SDU.

This is how OSI RM and TCP/IP RM, respectively, send data from one upper layer peer entity towards another upper layer entity via a lower layer communication network. If one compares the schematic model with figure 2.2 (page 32), the CEP is implicitly represented by the interface between the layer entity and the complex connector; and the SAP is implicitly represented by the interface between the converter component and the layer entity.

Be aware that figure 5.1 shows two different but complementary snapshots of a layered system. One snapshot is the layer (N) network model with its complex connector C . The other snapshot is the combined layer (N)/layer ($N - 1$) model with the complex connector C being removed and replaced by the lower layer network model. We will stick to this “two in one” drawing style also in subsequent diagrams.

Mathematical Model of Layered Distributed Communication

To further simplify the model, we aggregate the lower layer entity and its related adaptor. As a matter of fact, the adaptor A belongs logically to the upper layer. Aggregating the adaptor with the lower layer entity narrows the layer model down to PDU interfaces and hides SDU interfaces; this eases our considerations. The

aggregation component is labeled with a P. The letter has been chosen intentionally: usually, the peer components P' and P'' (more precisely, the components L'_{N-1} and L''_{N-1}) implement a communication *protocol*¹. Furthermore, we remove the SDU/PDU comments, which indicate the meaning of the timed stream, and replace them by more abstract names for the channels. Finally, we get rid off the layer (N) entities, since they are sufficiently abstracted by the interface of the complex connector. The resulting model is shown in figure 5.2.

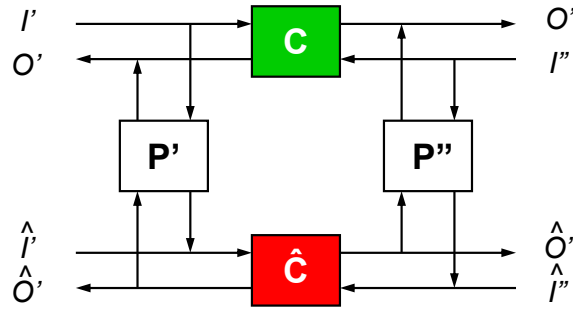


Figure 5.2: Mathematical model of layered distributed communication

The list of input channels represented by i'_C are of type I'_C , for the channels o''_C there is $o''_C : O''_C$ etc. The concatenation of the input channel lists i'_C and i''_C is denoted by i_C and is typed by I_C , the concatenation of the type lists I'_C and I''_C . The same conventions apply for the output channels of C and the input/output channels of \hat{C} . As introduced above, the complete list of channel/type pairs is abbreviated by I and O for C, and \hat{I} and \hat{O} for \hat{C} . I is composed of the concatenation of the lists of channel/type pairs $I = I' \frown I''$, with “ \frown ” being the concatenation operator. Analogously, $\hat{I} = \hat{I}' \frown \hat{I}''$, and so on.

At their level of abstraction, the syntactic interfaces of the specifications of the complex connectors C and \hat{C} can be defined as follows:

$$C \in (I' \triangleright O'') \cup (I'' \triangleright O')$$

$$\hat{C} \in (\hat{I}' \triangleright \hat{O}'') \cup (\hat{I}'' \triangleright \hat{O}')$$

Here, we assume that the complex connector decouples the direction of message transfer. For most cases, this is a sufficient approximation. In a more complicated model the input e.g. of any channel of I' might influence the output on any channel of O' (e.g. electromagnetic interference). The definition can be extended accordingly and is expressed quite densely by:

$$C \in (I \triangleright O) \tag{5.1}$$

¹Or *protocol protocol* according to chapter 2.

$$\hat{C} \in (\hat{I} \triangleright \hat{O}) \quad (5.2)$$

The protocol components P' and P'' *per se* have to realize a complex interwork: messages have to be processed at the connector interface and, concurrently, requests from or notifications to the upper layer have to be handled.

$$P' \in ((I' \frown \hat{O}') \triangleright (\hat{I}' \frown O')) \quad (5.3)$$

$$P'' \in ((I'' \frown \hat{O}'') \triangleright (\hat{I}'' \frown O'')) \quad (5.4)$$

The arrangement of P' , \hat{C} , and P'' is a form of composition we called *mutual feedback*, $P' \otimes C \otimes P''$.

The first important statement about the relation of layers we can formulate is that the complex connector C and the layer $(N-1)$ composition $P' \otimes \hat{C} \otimes P''$ share the same syntactic interface. Additionally, we have to put demands on a specific sort of semantic relationship which we introduced as *behavioral refinement*. The fact that the behavior of C can be substituted by the composite P' , \hat{C} , and P'' is now adequately formulated by the requirement

$$C \rightsquigarrow (P' \otimes \hat{C} \otimes P'') \quad (5.5)$$

Furthermore, the protocol specification given by $P' \otimes P''$ has to fulfill a specific requirement: any communication history fed in at the input of P' (I' in equation 5.3) is retrieved at the output of P'' (O'' in equation 5.4) and vice versa (input at I'' and output at O'). Only a time delay might occur. This requirement is expressed by the *identity relation*.

$$P' \otimes P'' = \text{Id} \quad (5.6)$$

The *identity* Id can be formally described by a component with two input channels a and b , two output channels a' and b' , and the requirement that

$$\bar{a} = \bar{a'} \quad \wedge \quad \bar{b} = \bar{b'} \quad (5.7)$$

The identity specification demands that any input stream is retrieved at the output neglecting time shifts. The overline stands for the untimed variant of a timed stream, where all ticks are simply being removed. Thereby, delays of messages become irrelevant.

The identity requirement is extremely minimalistic but condenses general design principles of communication protocols. Equation 5.6 and 5.7 state that messages must be conveyed orderly, i.e. in a deterministic manner, with respect to their interfaces towards the upper layer. Assumed that we principally cannot restore the chronological order of messages at the destination, the destination must have an infinite number of independent, conflict-free states to process messages. Such a machine is of no practical use in communications. Thinkable is only a protocol

that has a finite number of independent space states, between which chronological ordering plays no role. Though, within each state space message determinism is demanded. This is a theoretical option e.g. for control-oriented protocols such as the Gateway Control Protocol (GCP) [ITU00] and would require to formulate several identity relations for each independent protocol part. The practical use may be limited.

Some protocols offer the possibility to deliver data messages in an unordered fashion, see e.g. the Stream Transmission Control Protocol (SCTP) [SXM⁺00]. The “unordered” option is not in opposition to the identity condition. Despite unorderedly delivery, the message order is determined e.g. by inband sequence numbers. Message reordering is left to the protocol user part. Component A in figure 5.1 is responsible for that in our model and covered by the identity condition.

A basic design principle of communication protocols is message encapsulation: higher level PDUs are encapsulated in lower layer PDUs, see also figure 5.1. We do not formulate any requirements on that, because the identity relation enables but does not enforce PDU encapsulation; in fact, the identity relation describes a more generic form of protocol packaging, which includes package segmentation and compression.

To summarize, equations 5.1-5.6 above describe a mathematical model of layering; they form a set of requirements on four components, which – if fulfilled – state that C can be regarded as a complex connector of a communication layer and that C can be refined by P' , \hat{C} , and P'' . \hat{C} reflects the lower layer complex connector, P' and P'' realize a communication protocol fulfilling the identity relation. We learned about the importance of the complex connector as the link between distributed communication entities; an aspect that is not made clear by OSI RM or any other reference on layering. As we will shortly see, this insight has severe consequences on the abstraction techniques that can be used to model communication systems.

5.1.3 The Abstraction Hierarchy and Implications on Architecture Design

In this subsection, we investigate the main consequence of layering: that layering is about abstraction. We will explain why layering leads to an abstraction hierarchy. As a next step we point out the consequences of the abstraction hierarchy on the architecture design of communication networks. When the interfaces of a specification are syntactically and semantically preserved, it is a matter of perspective to interpret a refinement relationship from a black-box view or from a glass-box view. Applied on the fundamental refinement relationship of the complex connector to the “lower” communication network (equation 5.5) we will end

up with either a *node-centric* design view or a *network-centric* design view. Both views lead to completely different but related approaches on systems design.

The Abstraction Hierarchy

As was introduced, layers abstract levels of communication networks, each level being concerned with its own domain of functionality and service of peer communication. The layers are strictly separated; nevertheless, they are related in an abstraction hierarchy.

If we rewrite formula 5.5 using indices N , $N - 1$ etc. to mark their layer relation, the recursive nature of formula 5.5 gets much clearer.

$$C_N \rightsquigarrow (P'_{N-1} \otimes C_{N-1} \otimes P''_{N-1}) \quad (5.8)$$

Component C_{N-1} is refined by and thus can be substituted by the next layer $P'_{N-2} \otimes C_{N-2} \otimes P''_{N-2}$ and so on. The recursion can be interpreted as a hierarchy that relates an arbitrarily selected complex connector C_N to a chain of k ($k \leq N$) protocol entities completed by another complex connector C_{N-k} . The situation is illustrated in figure 5.3.

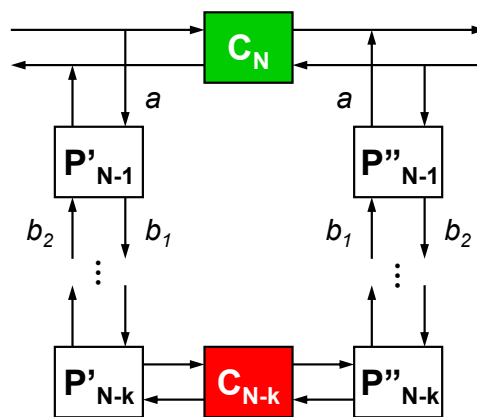


Figure 5.3: Layered model: the abstraction hierarchy

Why does figure 5.3 reflect an abstraction hierarchy? In other words, why is C_N more abstract than C_{N-k} and not the other way around? The answer is implicitly included in the identity condition of peer protocol entities. The equation $P'_N \otimes P''_N = \text{Id}$ describes an ideal transmission case, meaning that C_N is replaced by a “short circuit”. If a message (PDU) a gets injected at the upper (or outer) layer interface of P' , the identity relation guarantees retrieval of a at the upper layer interface of P'' . On the inner peer-to-peer communication interface, the protocol entities support another set of PDUs, say b_1, b_2 . Message a can be

only restored at the receiver's side, if the information to reconstruct a is transported via b -messages. Message fragmentation and compression might be used for that purpose. The same argumentation holds for PDU b_1 , if it gets delivered to the next lower protocol pair. What can be seen from this scenario is that more and more information is to be processed by the protocol entities deeper down in the layer hierarchy. While we send, for instance, a fax over a virtual call connection, hundreds of thousands of bits in form of electric pulses are actually transmitted over a copper cable. The involved chain of protocol layers uses data compression and redundancy mechanisms to ensure that message a , the fax, remains intact. The complex connector C_N (the call in our example) abstracts away all the lower level details of message processing but preserves the interface behavior. In that sense, the layered communication model in figure 5.3 represents an abstraction hierarchy. Notably, we provided a fully algebraic explanation for layered communication hierarchies.

The picture of an abstraction hierarchy might be a bit misleading, though. Communication layers are by no means more or less complex or complicated than other layers are, no matter of where they reside in an abstraction hierarchy. Each communication layer is a system expert domain of its own with its own dimension of networking problems and solutions. It is not necessarily the layer as such, which defines an inherently degree of abstraction, it is the assigned purpose of a communication layer in a stack. This explains, why the abstraction levels of different protocol stacks are hard to compare. That is the reason why the comparison of OSI RM and the Internet Architecture is notoriously problematic and difficult.

Node-Centric Design View

The principle of black-box refinement sets two different component arrangements into a *replacement* relationship ruled by some syntactical and semantical conditions on the interfaces. In our case it is a communication network that replaces the “upper” complex connector as long as the conditions described on page 162 ff. are fulfilled.

If we apply this principle on the abstraction hierarchy or – to simplify our considerations – on the component-oriented OSI model, see figure 5.1, the upper complex connector gets replaced by a distributed communication system; two piles of protocol components plus the “lower” complex connector remain, see figure 5.4. We group the protocol piles on each side in order to keep the simple model of an entity-connector-entity communication network, $P'_{N,N-1}$ on the left hand and $P''_{N,N-1}$ on the right hand side.

What we end up with is the typical OSI-like viewpoint on layered distributed systems: nodes of protocol stacks with the nodes connected by some communication link. This node-centric design approach has a long tradition in the telecommu-

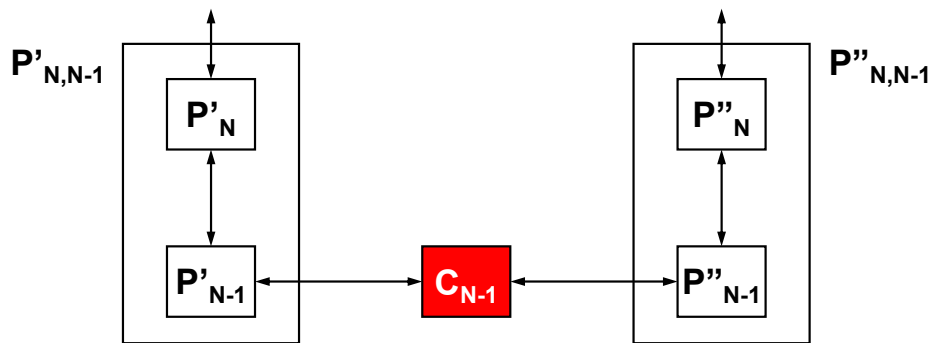


Figure 5.4: Node-centric design approach

nication community and is the dominating design paradigm for the architecture of communication systems. Network design is to a large extent equated with the design of protocol stacks. Diagrams of protocol stacks are typically used to present an overview of even rather complex systems such as GSM (Global System for Mobile Communication) or UMTS (Universal Mobile Telecommunications System), see e.g. [Wal01].

While there is nothing wrong with a node-centric design approach, OSI's dominance in that respect led to a depreciation of the connection concept. OSI's weak support on the notion of "virtual" connections and the implicit idea of the abstraction hierarchy (in the sense presented) gives rise to some criticism. The author noted that experts in the domain, if asked to locate the abstraction of a connection in a protocol stack diagram, notoriously fail to relate complex connectors to the right level in the protocol stack. For example, a complex TCP connector does not relate to the TCP protocol layer (the provider level), it is the user of TCP such as HTTP (Hypertext Transfer Protocol) [FGM⁺99] that uses a TCP connector as an abstraction of TCP. Such mistakes indicate that the concept of a complex connector as an abstraction tool is definitely not used. It is *not* common practice to cut off protocol stacks at an arbitrarily level and abstract away the "lower" parts in form of a complex connector and thereby shorten the protocol stack. By not doing so developers deprive themselves of a powerful design technique.

Network-Centric Design View

The white-box principle of refinement sets two different component arrangements into a *nesting* relationship. In our case it is the "upper" complex connector that hosts a complete communication network. As a result, all structural elements of protocol components and complex connector remain existent; it is just that "inner" networks are contained in the complex connector of the "outer" network, see figure 5.5. The complex connector hides an infrastructure to the outside, which it

unveils only by looking inside the connector.

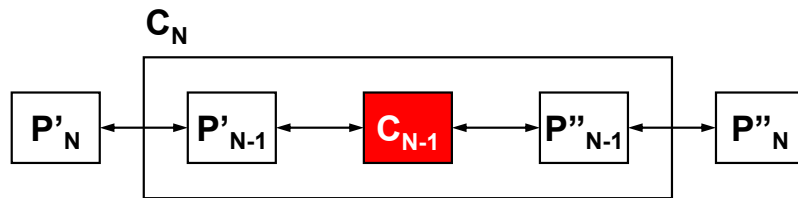


Figure 5.5: Network-centric design approach

It is amazing to note that a systematic use of the network-centric design approach is not established practice. There are almost no sources in literature that nesting networks is an option nor did we find any convincing evidence in industry. One rare example is SDL and its channel substructure concept [EHS97]; however, these techniques have not been evolved to a systematic methodology in network design. This is even more surprising, since a network-centric design approach is an ideal technique for architects who aim to engineer networks not protocol stacks. It enables the architect to stepwise resolve the granularity of a network architecture and precisely formulate the requirements on the next network layer by means of complex connectors.

Viewpoint Matters

If we look on the phases of system development, we see that both approaches are of relevance and help significantly improve design and test of communication systems. It is a very natural way of viewing communication systems from a network-centric standpoint in early design phases and to gradually shift view towards a node-centric standpoint in implementation phases. To paraphrase the key idea: communication systems are designed as networks but implemented as nodes.

The abstraction hierarchy enables system architects, designers, as well as testers to limit their focus of interest on a narrowed level of complexity of the respective view.

5.2 Layering in ROOM

We already gave a brief introduction to ROOM's layering concepts in section 2.3 (page 53 ff.). This section is based on that introduction but provides a more detailed view on and a thorough discussion of ROOM's layering conception.

5.2.1 Criticism on ROOM's Interlayer Model

According to ROOM, layering is a modeling principle that structures the organization of a system in the highest form. To describe layers, ROOM takes over the idea of OSI that all interaction between adjacent layers takes place through discrete points on the interlayer boundary, namely the SAP. ROOM applies the typed port concept to SAPs, thereby trying to harmonize interlayer and peer communication interactions. The OSI-SAP is divided into two interface components, a ROOM-SAP², which is attached to the upper(!) layer, and a *Service Provision Point* (SPP), which is attached to the lower layer. As a consequence, an actor has three categories of interfaces: ports represent peer interfaces, SPPs represent service provisioning interfaces, and finally ROOM-SAPs represent service usage interfaces. Together, these three interface categories define the type of an actor. Both the ROOM-SAP and the SPP are connected by a communication channel called *layer connection*. The ROOM-SAP is associated with a protocol P, and the SPP is by definition associated with the conjugated protocol P*, see figure 5.6. (Remember, the conjugated protocol P* has the same definition as P except that the incoming and outgoing message sets are interchanged [SGW94, p.154].) Neither the ROOM-SAP nor the SPP have an explicit graphical notation; the grey shaded boxes in figure 5.6 are shown for explanatory purposes only. The layer connection is symbolized by an arrow (with the arrowhead pointing to the "lower" layer) and represents a collection of individual but not-notated ROOM-SAP to SPP links.

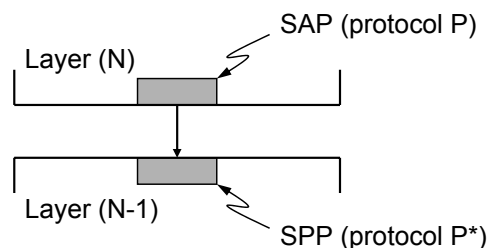


Figure 5.6: Layers in ROOM, see [SGW94, p.201]

²The prefix "ROOM" has been introduced to clearly separate OSI-SAPs from SAPs in ROOM, which are quite different.

Even though we can widely agree on ROOM's approach, we disagree on some details. There are three major criticisms on the interlayer modeling approach: first, ROOM introduces a confusing interpretation of the SAP concept by attaching it to the service user; second, ROOM misses an important semantic constraint implied by SAPs; and third, there is the lack of a proper notation for SAPs. The following paragraphs explain the points mentioned in more detail.

(By the way, it goes without much saying that ports and ROOM SPPs/SAPs are exclusive to each other: a port cannot be connected to a ROOM SPP/SAP.)

Confusing ROOM-SAP concept

As was elaborated on previously, an OSI-SAP belongs to the service provider and not to the service user. The lifetime of an OSI-SAP is bound to the lifetime of the service provider. In addition, any number of service users, who comply to the SAP interface, can be connected to the SAP of the service provider. This is exactly the converse of ROOM: The SPP in ROOM corresponds to the OSI-SAP. Nevertheless, the ROOM-SAP (on the service user side) specifies the interface protocol even though it is the service provider, which defines and publishes its service interfaces; that is also how technical standards are describing interfaces. ROOM does it the other way around and shows consistency with its recommendation to define protocols from a client's perspective for peer-to-peer communication. However, this leads to further confusion in the SAP case. Properties like QoS and SAPI can not be sensefully associated to the ROOM-SAP. Only a service provider can guarantee specific QoS and be addressed by an SAPI. However, the SPP is not prepared to catch such properties either. In short, attaching a well-known concept like the OSI-SAP to a service user or simply naming the interface component of a service user "SAP" does not only confuse people familiar with telecommunications, it is objectively wrong.

Insufficient semantic preciseness

SAPs are a port-like concept, but they are semantically different. The similarity is that SAPs and ports can (but do not need to) be based on an homogeneous communication model of interaction (e.g. message based protocols) [SGW94, p.200]. The difference is that SAPs are a structuring measure, they organize collectives of actors communicating via ports into layers. That means that we have to introduce a semantic rule saying that if two peer-to-peer networks are separated by an ROOM-SPP/SAP pair, none of the actors is allowed to be connected to another actor of the other network via ports. In other words, peer communication should not bridge layers. The ROOM-SPP/SAP concept constraints how actors may communicate to each other via ports. It is this constraint that semantically introduces

peer-to-peer and interlayer communication.

In ROOM, it is possible to violate that semantic rule, see for example figure 5.7. In that figure a coordinator capsule is connected via ports to a layer 3, 2, and 1 capsule; the coordinator via peer-relations bridges the layer connection of the layer capsules. The reason for violation is that such semantics are not embedded in the formal specification of the abstract syntax of ROOM, see [SGW94, p.212ff.]. Maybe the lack of a plane concept (this is introduced in section 5.4) enforced the authors to be less restrictive. SAPs/SPPs in ROOM prove to be a qualifying concept, which *can* be used for the semantical intention of modeling layers. The deficiency in semantics does not clearly separate port connections from layer connections. Strictly speaking, ROOM fails with its claim that with its support for layering, “layered system architectures become not only more explicit and semantically precise, but also enforceable” [SGW94, p.194].

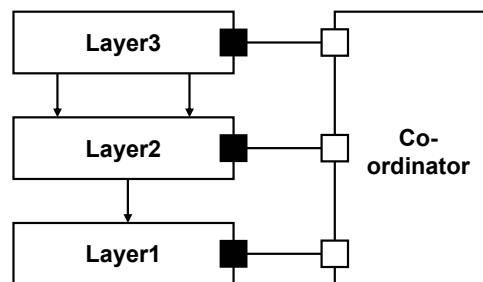


Figure 5.7: Coordinator capsule bridging layers, see [SGW94, p.208]

On the other side, ROOM is very strict on layering. No upper layer is allowed to access any layer that is below its adjacent lower layer. In practice, system engineers are wary of layering violations, but there are some circumstances in which it is required to do so [Kes97, p.71f.]. In that sense, the SAP concept is less restrictive than layering is; there is no “upper” or “lower” layer but the roles of service providers and users in any sort of arrangement. We see this as a benefit, others might interpret it as a sacrifice to the imperfect world of design.

Lack of proper notation

ROOM argues that in most applications the number of individual interlayer connections tends to be very large. That is why only an “embracing” notation, the *layer connection* symbol, is introduced; a notational symbol for ROOM-SAP/SPP connections is regarded as superfluous [SGW94, p.201f.]. We do not follow this argumentation. We believe it to be meaningful to have a notation at hand for individual ROOM-SAPs and SPPs and their relationships. A modeler should be equipped with this option as well. A suitable notation is proposed subsequently.

The discussion and commentary on ROOM should be regarded as an improvement proposal for the language. For clarification purposes note that SAPs (as introduced above) are not just an extended version of ports. The SAP and the port concept are rather specializations of the same base concept, which one could for example call “boundary interface concept”. In contrast to the port concept, the SAP concept additionally has attributes (QoS, SAPI) and puts semantic constraints on the use of ports. Since we would like to promote SAPs in the software engineering domain as well, we do not demand strict OSI compliance and declare the SAP attributes as optional.

5.2.2 Improvements to ROOM’s Interlayer Model

Two of the problems mentioned are easy to solve: To avoid confusions about the ROOM-SAP and ROOM-SPP concept, we introduce slightly different conventions about naming and message schema conjugation. While this is just a syntactical issue, the semantics of ROOM’s interlayer model are not impacted. Similarly, the introduction of a proper notation for interlayer details is uncritical to the ROOM language. The remaining issue, insufficient semantic preciseness, is something we need to discuss in further detail.

Improved Conventions

To avoid all confusions around the term “Service Access Point”, we drop it completely. We agree on the following conventions: the layer interface attached to the service provider is called *Service Provisioning Point* (SPP); so far, nothing has changed. The opposite layer interface attached to the service user is called *Service Using Point* (SUP) instead of Service Access Point (SAP). By convention, we will always define the message schema from the viewpoint of the SPP; the SUP holds the conjugated message schema. If required, the SPP can be typed by a data class. The data class represents the *Service Access Point Identifier* (SAPI) or, according to the new terminology, the *SPP Identifier* (SPPI). As we learned in the previous chapter, the SPPI can be an IP address, an TCP/UDP port or anything else appropriate.

From now on, we will use the new terminology and talk about SPPs and SUPs. Remind yourself that this is just a terminology issues and not a language change. The reader can always map back the SPP to a ROOM-SPP and the SUP to a ROOM-SAP and swap the message schema conjugations, if needs be. Readers, who are still in favor of a SAP concept in the OSI sense, may regard the SPP and the SUP as a conceptual unit representing the OSI-SAP.

So far, a layer connection is not very much different from a binding. The difference only is that there are defined naming conventions for the interfaces involved

in a layer connection: SPPs correspond to “black” ports, SUPs to “white” ports. Otherwise, SPP and SUP also have a reference name and a replication factor. Just the conjugation indicator has become superfluous because it is predefined for SPP/SUP.

Improved Notation

Due to their similarity, we graphically notate SPPs and SUPs like ports, but use a diamond symbol instead: The SPP is visualized by a “filled” diamond symbol, the SUP is visualized by an “empty” diamond symbol. The layer connection between the SPP and the SUP is visualized by a line like a binding contract. Figure 5.8 is a re-drawing of figure 5.6 using the improved notation.

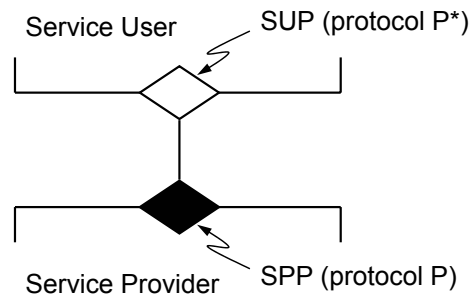


Figure 5.8: Notation for SPPs, SUPs and layer contracts

If an actor class is nested inside another actor class via an actor reference (see figure 2.14 on page 58), the SPP and the SUP are treated in a special way: SPPs can be *selectively* exported to the container actor class via an export connection, whereas unfulfilled (i.e. unbound) SUPs are exported *automatically* and become part of the container actor class’ interface.

Selectively exporting SPPs is semantically the same as connecting the “internal” SPP to an “external” relay SPP. We will notate SPP relaying in exactly the same way as we do relaying for ports, see figure 5.9. One SPP is exported, the other is not.

For SUPs it does no harm to ROOM if we demand manual export of unfulfilled SUPs and leave it to the model checker to validate if *all* SUPs have been relayed inside-out or bound to an SPP. The effect is the same as for automatic SUP export, we just prefer the “make it explicit” style, see figure 5.9. You can see this diagram as an illustration of figure 2.14 (page 58) in explicit notation.

It is a matter of the presentation level to explicitly show SPPs and SUPs, their exports and interconnections or to change the granularity of visual information, remove the SPPs and SUPs symbols and show layer connection symbols (the arrow)

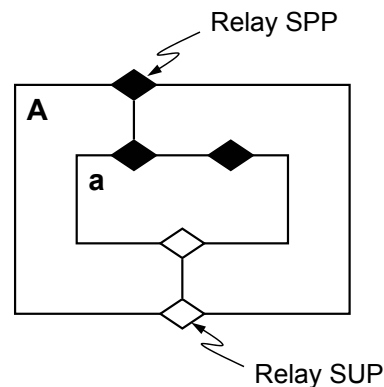


Figure 5.9: Explicit export notation for SPPs and SUPs

only. The point is that we specify our models explicitly without any ambiguity and leave it to a computer aided tool, to change the presentation level. Again, this is no essential change to ROOM but just an issue of the visual specification philosophy.

Semantic preciseness

Peers of ports and SPP/SUP pairs seem to be very similar and conceptionally close, especially since we harmonized the notational conventions. So, where is the difference?

The semantic difference between layer interfaces (SPP, SUP) and peer interfaces (ports) is subtle but important. A port, no matter if conjugated or not, *can* be but *must not* be bound to another port (external end port or relay port) or the behavior component (internal end port). Ports impose no artificial constraints. Not so SUPs. An SUP of an actor reference *can not* be connected to the behavior component of the composing actor class but *must* be exported or bound to an SPP. That means, an actor having a SUP communicates via the SUP to another actor that is outside the scope of the former actor's nesting context. Nesting in ROOM is strict containment, so nesting is related to the lifetime context of an actor. If an actor is destroyed all its containments are also destroyed. In other words, an actor having a SUP communicates via the SUP to an actor in another lifetime context. SUPs have a structural and a run-time implication.

It is this simple rule (“SUPs must be exported or bound to complementing SPPs”) that basically distinguishes a port from a SUP. And it is this simple rule that is claimed to be at the heart of the notion of layering – at least if we follow ROOM's approach. The consequence is that SUPs enforce a “higher” structural organization than ports do. SUPs separate nesting contexts or, better said, containment contexts. To quote SELIC et al. [SGW94, p.195]:

The principal difference [to containment] is that, in layering, the upper layer does *not* contain the lower one.

This is a fairly reasonable condition for layers, but possibly not a sufficient condition. In addition, SELIC et al. say [SGW94, p.194]:

Layering is a strictly *hierarchical* relationship. An upper layer's implementation depends on the lower layer (but not the other way around).

By definition, the actor class having the SUP of a layer connection is said to be the “upper” layer, the actor class holding the SPP is said to be the “lower” layer. Unfortunately, the authors of the ROOM book do not install further semantics for the notion of “depends on”. Two actors bound together via a SPP/SUP layer connection are not more or less dependent from each other than two actors via a binding contract are. It remains unclear, on what a “hierarchical relationship” is based upon.

Here, we can help out. In chapter 3, “Types of Communication”, we introduced the notion of control, derived three basic types of communication relationship (control-oriented, protocol-oriented and data-oriented) and proposed a notation. This classification scheme can be taken over for SPPs and SUPs. An SPP can be control-oriented or data-oriented, so can be the SUP. If we by definition say that it is never the SUP that solely exerts control, we can distinguish three variants of layering, see figure 5.10. Many people equate layering with control-oriented layering, but that is an arbitrary limitation. The notation is aligned with the notation used in chapter 3.

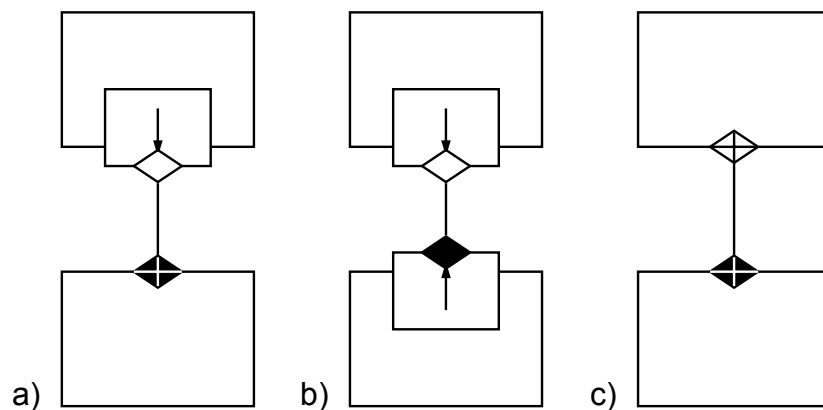


Figure 5.10: Communication types between layers: (a) control-oriented, (b) protocol-oriented, (c) data-oriented

In ROOM, layering is a synonym for the *vertical* direction of communication. This naming convention is common. Usually, the communication between layers

is assumed to be ideal, so case (b) in figure 5.10 is of little practical relevance, but still it is an option. This said, the basic patterns for vertical interaction in a communication system shown in figure 3.24 (page 111) still hold valid. We just have to apply the new notation and replace ports by SPPs and SUPs in order to indicate vertical direction of communication, see figure 5.11. Meanwhile, we have learned that there is no need to connect the data port of the Connection directly to the “lower” layer; one can use the import feature of ROOM to plug-in the service user in another context for data-oriented transfer. For that very reason, we leave the data port of the Connection unbound in figure 5.11

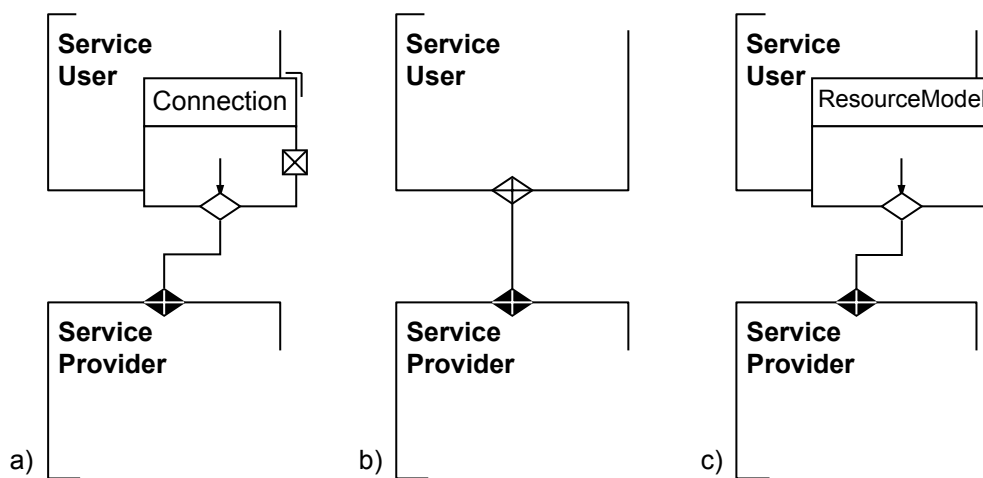


Figure 5.11: Patterns of vertical communication in a communication system: (a) connection-oriented, (b) connectionless, and (c) resource control services

Layering Semantics

As was mentioned, on the one hand, ROOM is very strict on layering: No upper layer is allowed to connect to any layer that is below its adjacent lower layer. On the other hand, ROOM is very relaxed on layering: Bindings via ports may break the rule of strict layering. Both aspects are summarized in figure 5.7.

We criticised both aspects as inappropriate for modeling (tele)communication systems. In contrast to ROOM, we will not insist on strict layering but demand (yet not enforce for the sake of flexibility) that there is no chain of bindings that principally enables one layer to communicate with another layer via ports. That is the real reason why we do not permit the data port of the service user in figure 5.11 a) being directly connected to the service provider. Importing the whole “upper” layer or a part of the upper layer inside a lower layer puts the upper layer (part) in an additional run-time context; but that does not impact the composition and life-time constraints we identified as key for layering.

In comparison to figure 5.7 we summarize the new layering semantics in figure 5.12. We intentionally have drawn the layered architecture inside another actor class to indicate that layering can appear on any level of granularity: on a system architecture level as well as on a “microarchitecture” level.

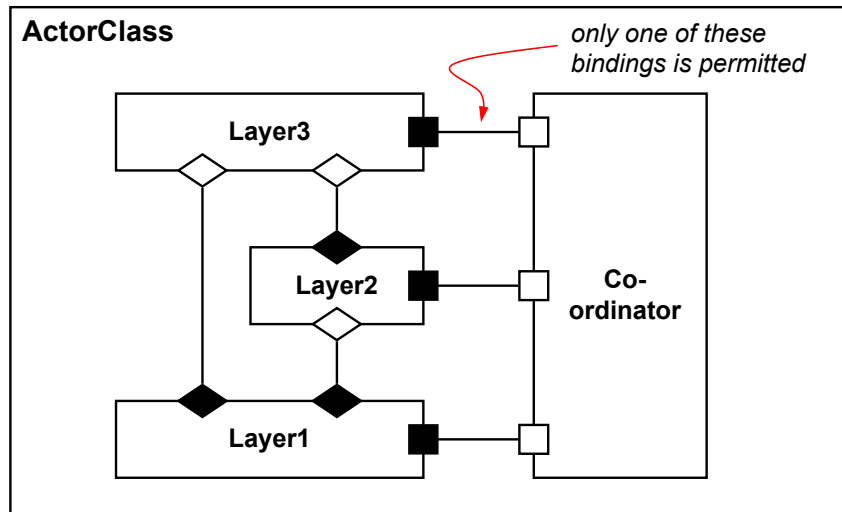


Figure 5.12: The semantics of layering exemplified

Beyond doubt, there are cases in which we need to “bridge” layers similarly like the Coordinator actor class does. But if we do not promote the use of ports between layers, we need something else instead. In telecommunications, this problem has been solved by the notion of *planes*. A plane is a structuring means that embraces a consistent set of layers. We could structure figure 5.12 into two planes: one plane consists of the layer set Layer3, Layer2 and Layer1; the other plane consists of the Coordinator as the upper layer and the conglomerate of Layer3 to Layer1 as the lower layer. We will introduce planes as an extension to layers in section 5.4.

5.2.3 Layering versus Layering

How is ROOM’s understanding of layering related to the algebraic model of layering we presented at the very beginning of this chapter? Not at all, to be tough; a little, to be modest and academically correct.

Layering as an Usage Relation

When we talked about SPPs and SUPs, we agreed on the terminology to say that the actor class having the SUP of a layer connection is said to be the “upper” layer,

the actor class holding the SPP is said to be the “lower” layer. Sometimes we use role names instead and call the “upper” layer *service user* and the “lower” layer *service provider*. Be aware that these are just naming conventions. It is important to realize that upper/lower and service provider/user are by no means indications for levels of abstraction. As CLEMENTS correctly points out in [CBB⁺03, p.95ff.], layering is not about abstraction. Layering is a special kind of usage relationship, which can be characterized by *allowed-to-use*. Each layer is knowledgeable about its own service domain but completely ignorant of other service domains. There is nothing that makes one layer an abstraction of another layer; on the opposite, two layers complement each other in an elegant symmetry [CBB⁺03, p.96]. An *allowed-to-use* relation imposes a strict or partial ordering, not more and not less. Layering is definitely not about abstraction, rather about separation of concerns.

A similar comment goes for so-called multi-tiered architectures [TvS02]. Multi-tiered architectures slice a system architecture in application or service domains, e.g. in a database, a GUI client, and a server, and are very common for business and enterprise applications. Still, it is a kind of “use” relationship that ties the parts together.

Layering as Communication Refinement

When telecommunication engineers talk about layering, they mean something completely different. Their notion of layering is algebraically captured in the mathematical formalism above. Layering in communication systems is a matter of *refinement*, or more accurately a matter of *behavioral refinement* as defined in definition 2.3 on page 67. Since we added an important requirement to behavioral refinement for communication systems, namely the identity relation (formula 5.6), we prefer to call this sort of refinement *communication refinement*.

Communication refinement truly is about abstraction in a system as was elaborated on in section 5.1.3. More precisely, the abstraction refers to the complex connector and not to the communicating entities of a communication network. A complex connector is an abstraction for a communication service that can be detailed (refined) by a complete communication network.

How Communication Refinement relates to Usage Layering

Communication refinement and its relation to the notion of layering in the ROOM sense becomes clear, when one looks at the process of stepwise refinement, relating the abstract snapshot to the refined snapshot. Refining a complex connector (abstract snapshot) leads to a cascade of communicating peers and another complex connector at the “bottom” (refined or concrete snapshot), see figure 5.3. The interesting observation is that the communication protocols P'_{N-1} and P''_{N-1} of one

communication network level are in a usage relationship to the next “lower” communication protocol pair P'_{N-2} and P''_{N-2} . Previously, the communicating protocol peers were *using* a complex connector; now they are *using* a pair of protocol entities.

This usage relationship corresponds to the aforementioned *allowed-to-use* relationship. It also corresponds to what we denoted by *vertical communication*. It is all the same. Consequently, we use SPPs and SUPs between communicating entities of different network levels when the abstraction of a complex connector is resolved.

The reader may now understand why figures that show stacked communication layers in a way figure 2.4 (page 37) does are totally misleading if not wrong. The transport layer is not a layer on top of the network layer, because the transport layer is not just *allowed-to-use* services of the network layer. We have to be more accurate and precise and say that the communicating entities of the transport layer can be layered atop communicating entities of the network layer, provided that the complex connector of the transport layer is refined to elements of the network layer. As we can see, the term *layer* is heavily overloaded, and one has to be careful with its use. That is why we try to avoid speaking of a transport or network layer and say transport network and network network instead.

Communication Refinement in ROOM

The notion of refinement is to some extent covered by ROOM’s capability to mark an actor class as substitutable. However, the notion of communication refinement cannot be adequately expressed in ROOM, even not by an extension to the ROOM language, since the identity relation implies constraints on the data flow in a model. ROOM is not an appropriate language that can reason on such a low level of detail. If important, one may choose another language and another tool, preferable a language which is based on an algebraic foundation like FOCUS [BS01]. From an engineering point of view, such reasoning is not necessarily required. Simply speaking, the identity relation equates to the demand of using protocols in a communication system. This demand is, so to speak, by definition fulfilled in communication system design. So, communication refinement is nothing we need to be notationally worried about. More vital is that system architects learn to think in abstract and concrete snapshots and relate both views. We regard this as a methodology issue.

5.2.4 Node-Centric and Network-Centric Designs in ROOM

We will end this section about layering in ROOM with a brief note on node- and network-centric architecture designs in ROOM. Both ways of utilizing communi-

cation refinement have different consequences on the architectural design.

Node-Centric Design in ROOM

Figure 5.13 shows how communication refinement is resolved with a node-centric design approach in mind. The complex connector C_N between P'_N and P''_N is refined by a communication network consisting of two “lower” communicating protocol entities, P'_{N-1} and P''_{N-1} , and C_{N-1} . If node-centric, the actor pair on the left hand side and the actor pair on the right hand side are embedded in the context of another actor each. Consequently, layering (now referring to layering in the ROOM sense) is inside the actors $P'_{N,N-1}$ and $P''_{N,N-1}$, which are also called *nodes*. Each node hosts a stack of protocols. On a coarse granular level, we see the same schema again: two entities (here nodes) communicating via some sort of complex connector.

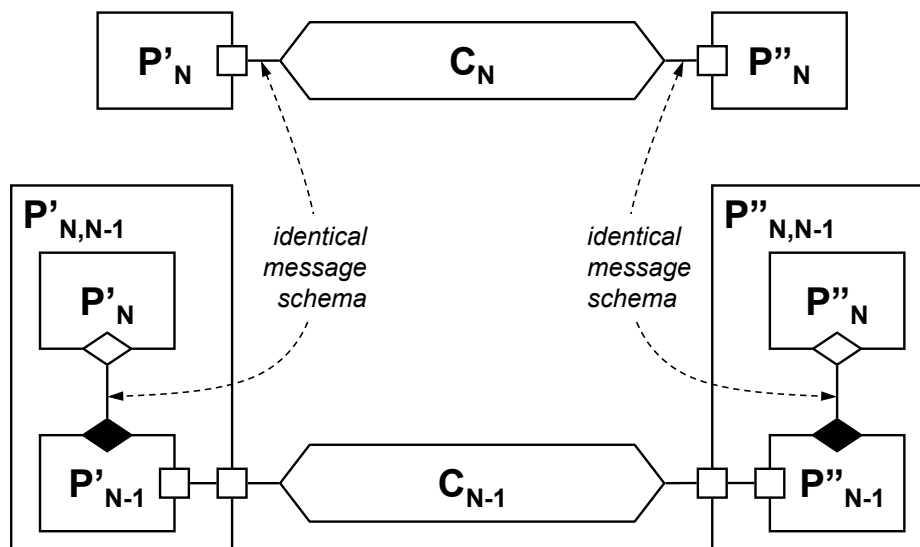


Figure 5.13: Node-centric communication refinement in ROOM++

The most striking observation in comparison to figure 5.4 is that we have a change in the kind of interface: a node-centric architecture style turns horizontal interfaces of remote peer communication into vertical interfaces inside nodes. That means, the architecture semantics of the interface change but not its syntax. The lesson to learn is that such transformations may change the architectural meaning of interfaces.

Network-Centric Design in ROOM

If communication refinement is resolved with a network-centric design approach in mind (see also figure 5.5), the interna of the complex connector C_N are replaced by a communication network, see figure 5.14. This is also called *implementation refinement*. In [CBB⁺03, p.192] the definition is: “Implementation refinement is a refinement in which many or all the elements and relations are replaced by new, typically more implementation specific, elements and relations.” Here, we have to read the word “implementation” as “architecture implementation”. The previous abstraction of C_N is made concrete; this concretization still follows the schema of two entities communicating via a complex connector.

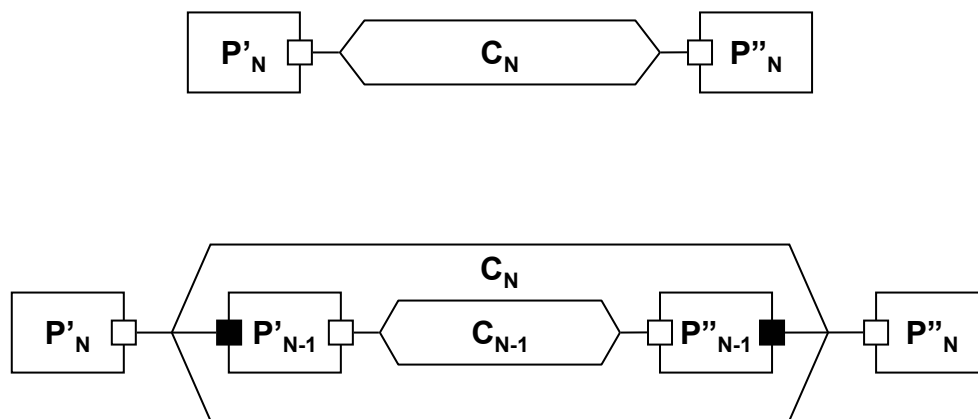


Figure 5.14: Network-centric communication refinement in ROOM++

In network-centric design, horizontal communication is preserved. Complex connectors are abstractions for complete networks, which may be made concrete.

Conclusion

As we can see, the simple pattern of two or more communicating entities, which are connected via a complex connector, is the reiterating schema in the architecture design of communication systems. The whole point of system architecture design is to resolve abstractions of complex connectors by suitable concretizations of communication networks. With the approach presented, modeling communication systems is effectively based on a simple but powerful abstraction technique. And that is how it should be. The system architect should focus on the technical domain and solve technical problems – that is what the architect is paid for and should spend most of his or her time on. Our approach makes it easy to introduce abstractions wherever needed in a system model, so that the architect can really

concentrate on the vital aspects of concern and express these with the features provided by ROOM.

We do not want to express any preference for the node-centric or the network-centric viewpoint on system architectures. Both are important. Therefore, in the next section, we just show how communication networks can be related to each other via the complex connector for the purpose of communication refinement, but leave the choice to the system architect to go for a node-centric or a network-centric approach.

5.3 Layered Communication Networks

In the following we demonstrate how communication networks can be related to each other by communication refinement. That means, we take the results from the previous chapter, in which we looked at each communication network as an independent, self-contained unit, and set it into relation to another independent, self-contained communication network.

5.3.1 Layered Networks exemplified on TCP

For TCP, we set the TCP *user* communication network, see figure 4.19 on page 151, into relation to the TCP *provider* communication network, see figure 4.22 on page 154. The basic approach is straight forward: The complex connector TCPUComService is removed and substituted by the TCP provider network.³ The result is shown in figure 5.15.

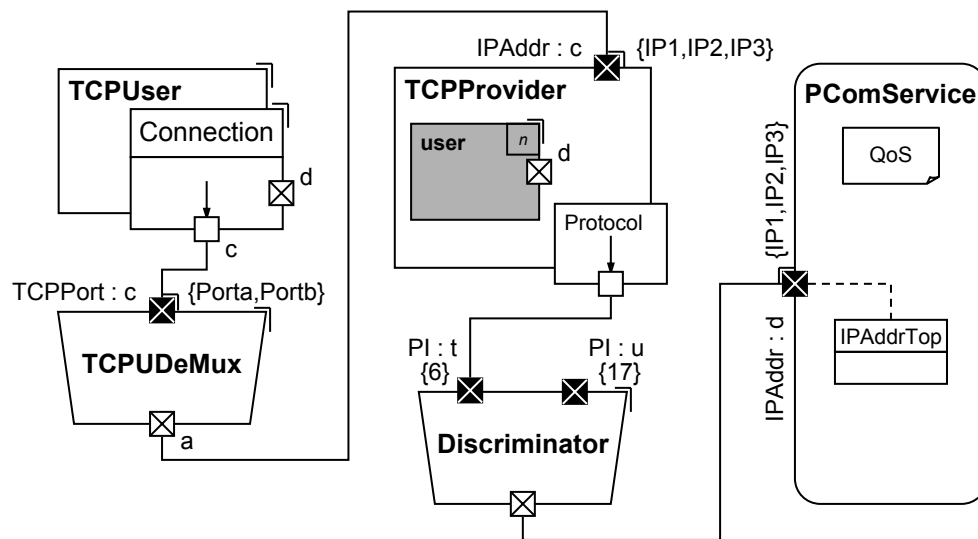


Figure 5.15: The TCP user network and the TCP provider network set into relation

There remains little to say about the diagram, since communication refinement is almost a mechanical process of substituting one by another as long as the interface syntax is preserved. Though, two minor issues may be worth mentioning: the change of TCPProvider and the “relaying” of the IP address space.

³Now we can see that it makes sense to regard the **TCPUComService** as a complex connector from a methodology point of view. **TCPUComService** is a very good example that a complex connector may be decomposed into other complex connectors detailing the functioning of communication.

Remarks about the TCP Provider

The actor class `TCPProvider` has experienced a slight change: Port `d` has been moved inside the actor class and is now attached to the imported actor reference `user`. There are two reasons for this change: (1) To enable the option for a node-centric grouping of actor classes, we have to prepare the model for layering in the ROOM sense. As we discussed previously, putting port `d` of `TCPUser` into another context of use via an actor import is the most elegant solution to preserve layering semantics of ROOM. (2) The actor import is also the most expressive means to show that the `TCPUser` has on one hand a control-oriented relationship (relayed by the demux) towards the `TCPProvider`, but is put on the other hand in a data-oriented communication context by the `TCPProvider`. This is what the architecture diagram says and that is what actually happens; it would be a severe loss of architectural relevant information if one would not use this powerful means of expressing processing relations.

Remarks about the IP Address Space

Another interesting observation is that the IP address space at port `c` of `TCPProvider` has “survived” the substitution process. As a matter of fact, communication refinement does not impact address spaces, though it might be an unexpected effect many system architects of communication systems are not aware of. The `TCPProvider` is addressed via a cascade of IP address and Protocol Identifier (PI). However, from the viewpoint of a `TCPProvider`, the `TCPUser` is identified and located via an IP address and a TCP port number. Of course, there is a relation between the IP addresses, which is established during registration procedures at the start-up phase. Usually, the `TCPProvider` “sees” the user under the same IP address as it has been addressed by the `PComService`. We may call this structuring of addresses *address relaying*.

5.3.2 Layered Networks exemplified on UDP

For UDP, the situation looks very similar. We set the UDP user communication network, see figure 4.21 on page 153, into relation to the UDP provider communication network, see figure 4.22 on page 154. The result of communication refinement is shown in figure 5.16 without further commentary.

Actually, the diagrams in figure 5.16 and figure 5.15 could be also be combined in a single diagram; it is just the protocol identifier, which discriminates the TCP part and the UDP part.

Once getting accustomed to reading ROOM diagrams including our extensions, it is quite much architectural information we supply on a visual level. This

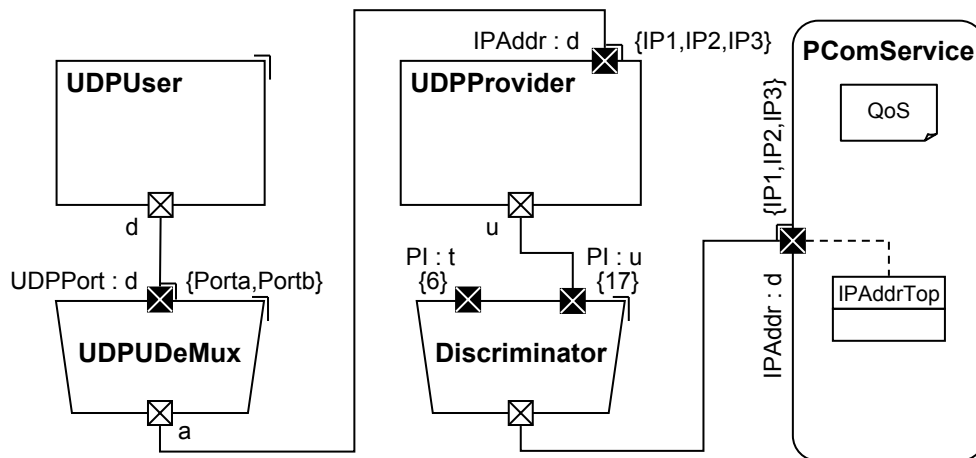


Figure 5.16: The UDP user network and the UDP provider network set into relation

is very much in contrast to raw box-and-line diagrams in SDL and other ADLs. Our approach also reveals the power of abstraction in form of complex connectors. Diagrams such as in figure 5.16 and figure 5.15 are hard to develop from scratch; but it is significantly simpler to develop the service user and the service provider part independently before setting them in relation of communication refinement. In addition, complex connectors allow the system architect to cut off the level of resolution at any point of his or her will.

5.3.3 Layered Networks exemplified on MGCP

Up to now, our models primarily focus on the aspect of distribution and layering (communication refinement) for connection-oriented and connectionless communication services. However, latest since the advent of UMTS and the physical split of the control and user plane (see also chapter 1) resource control can be also subject to distribution and – in consequence – also subject to layering.

That means, horizontal communication can be distributed in the same manner as vertical communication can be. In effect, we can use the instrument of a complex connector disregard of any “direction” of communication. That does not only concern resource control but also vertical service interfaces like e.g. the TCP/UDP service interface between user and provider. In this section we will exemplify modeling of remote vertical communication on MGCP, which also leads to a new interpretation of what an complex connector is good for: for modeling aspects in a system.

We discussed the Media Gateway Control Protocol (MGCP) in chapter 3 and presented a model in figure 3.22 on page 108. The first step is to replace the ideal

binding between MGC and MG by a fixed complex connector (a typed binding in ROOM) with adequate QoS attributes, see figure 5.17. Since MGCP expects reliable transfer of transaction messages, the complex connector should behave almost as ideal as the ideal binding does.

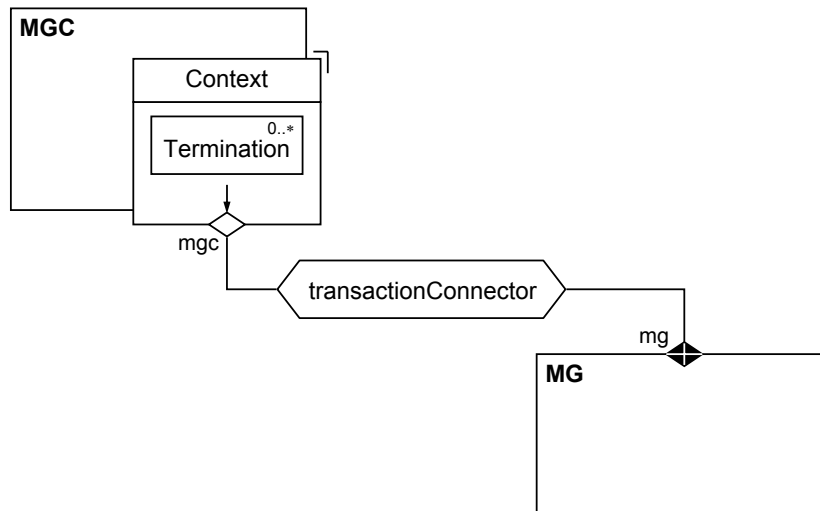


Figure 5.17: Model of a distributed resource control relation

If we want to fall back to proven technologies of mastering distant communication, one possibility is to base the refinement of the complex connector on TCP, see annex D of [CGH⁺00]. This case is shown in figure 5.18. Note that the actor classes `MGCCConnMgmt` and `MGConnMgmt` are only two specializations of a TCP user as we know it from the previous chapter, see figure 4.19 on page 151. The connection managers host some complex functions like e.g. protocol version negotiation and failure procedures. For the sake of a more compact drawing, we did not cascade the address space of IP addresses and TCP ports but created a combined address type `IPPortAddr` that contains an IP address and a TCP port as tuples.

What have we done here? First, we just inserted a complex connector between the MGC and the MG. From a MGC and MG perspective, this insertion has no observable effect, since the `transactionConnector` is supposed to be an almost ideal connection. Then we proposed a communication refinement for the complex connector. The refinement consists of two connection management functions, which behave like two TCP users that make use of a `TCPComService` and send and receive transaction requests and replies from the MGC and the MG, respectively. We did not resolve the `transactionConnector` by a completely new communication network, but added missing functionality in order to make use of a `tcpConnector`. If we interpret the refinement from a node-centric viewpoint, then the MGC node

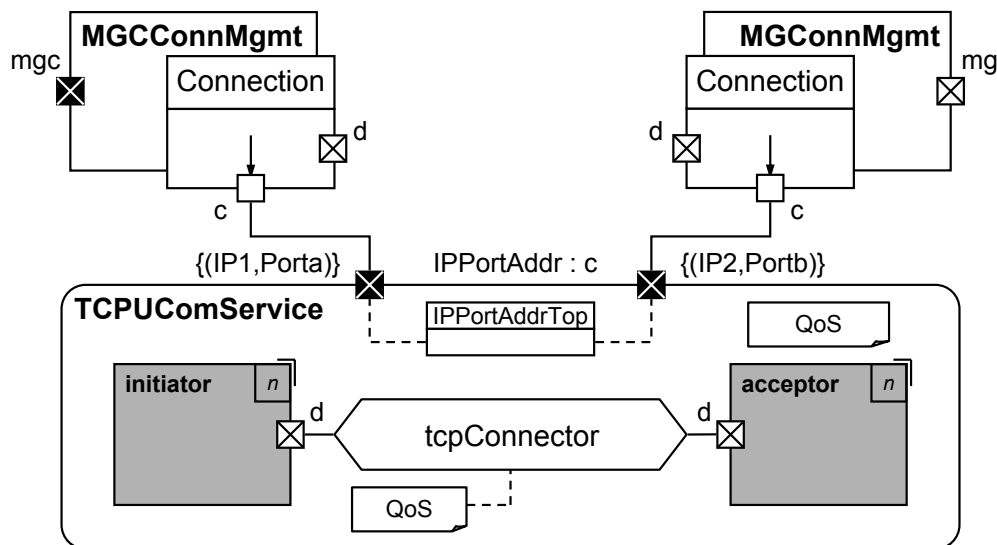


Figure 5.18: Refinement of complex connector between MGC and MG

and the MG node become enriched by connection management functionality. For the MGC node, see figure 5.19.

Adding missing functionality by communication refinement is a way of adding a *concern* or an *aspect* from a node-centric viewpoint. The use of complex connectors goes beyond the stratification of a communication system into layers of well-defined communication networks given by a reference model. With complex connectors, a system can be gradually decomposed in strata of concerns, each stratum adding a certain aspect to the system architecture. Complex connectors are a very powerful tool for abstraction and separation of concern purposes. In other fields, researchers also start to understand the usefulness of complex connectors, see e.g. [AK03]. More information about this can be found in chapter 8.

Conclusion: Even though the complex connector was motivated as a means to model distribution, we can generalize its purpose to model any communication relation and use communication refinement to stratify different sorts of aspects of that relation.

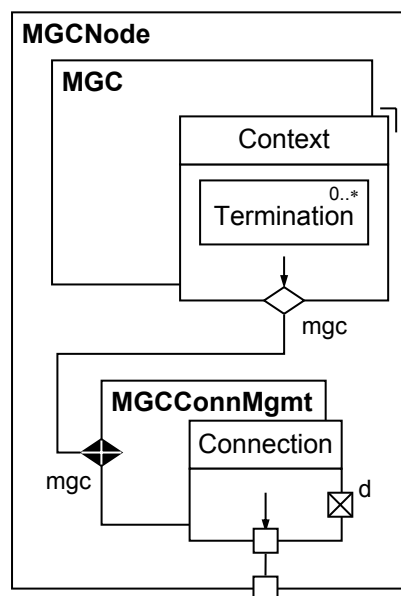


Figure 5.19: The MGC node

5.4 Planes in Communication Networks

Even though telecommunication standards very often refer to OSI RM, many of them add concepts and principles of their own; mostly, in order to structure the problem/solution domain for increased understanding and to provide intellectual tools for tackling the problem at hand (which is a sufficiently precise specification of a technical system for a specific purpose). Via this process of introducing, refining, and dismissing concepts in new standards, the telecommunication domain as such constantly evolves. Unfortunately, there is neither an authoritative nor an informal source of information that documents the current set of the most popular concepts and principles.

One of the concepts that has survived over the years and turned out to be extremely useful is the concept of *planes*. The concept was introduced in ISDN (Integrated Services Digital Network) [ITU93a], taken over in GSM (Global System for Mobile communication) [EV98], and currently shapes the network architecture of UMTS (Universal Mobile Telecommunications System) [WAS01]. The distinction is usually in three planes, namely the *control plane*, the *user plane*, and the *management plane*. Figure 5.20 shows a simplified version of the GSM plane architecture, which is almost identical to the ISDN architecture.

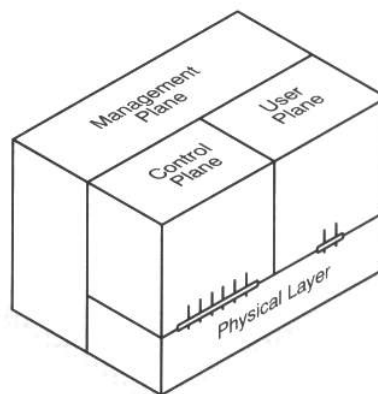


Figure 5.20: Informal model of the ISDN/GSM system architecture; taken from [EV98, p.117]

5.4.1 The Plane Concept in Telecommunications

A plane encapsulates service functionality and may have internally a layered (protocol) structure. Planes are an organizational means on top of layering and communication refinement. In telecommunications, the *user plane* provides for user

information flow transfer (data PDUs), along with associated controls (e.g. flow control, recovery from errors); the *control plane* performs call and connection control functions (control PDUs), dealing with the necessary signalling to set up, supervise, and release calls and connections; the *management plane* takes care of (a) plane management functions related to the system as a whole including plane coordination and (b) functions related to resources and parameters residing in the layers of the control and/or user plane [ITU91]. Figure 5.20 displays the relation of the planes in a three-dimensional arrangement; the *physical layer* is shared by the control and user plane for transmission purposes.

OSI RM is not prepared to handle planes (TCP/IP RM is not as well), which is also one of its major deficiencies. The control and user plane are not separated, the control and user information in one network layer always maps to user plane information of the layer below. Most OSI protocols do not provide facility negotiation during the active phase of a call, meaning that the system either works in user or control mode but cannot support signalling in parallel to user data transmission [ITU93a, chap.5.2].

The lack of a formal plane concept very often leads to diagrams similar to figure 5.20. Such diagrams provide at best an informal view on the architectural conception rather than a precise architectural model. On a case by case basis, designers had and still have to invent individual solutions in order to handle planes in their models. For example, in ISDN the engineers introduced a *Synchronization and Coordination Function (SCF)* as a major component of the management plane. The SCF is connected to the highest layer of the user plane and to the highest layer of the control plane via ordinary SAPs to coordinate and synchronize the required collaboration of planes [ITU93a]. This solution does not seem to be very ingenious; rather, it seems to be very specific and does not really address the concept of planes. However, a generalization of this approach unveils its full power, even though the designers of ISDN might not have ever thought about it.

5.4.2 Introducing Planes in ROOM

The idea is simple: If we add means to distinguish ports or SPPs/SUPs from each other, we are also capable of distinguishing planes. A classification of ports and SPP/SUP pairs by attributing them with a plane tag allows us to clearly identify a port, SPP or SUP as a member of a set of interfaces, which is just another paraphrase for planes. The idea is intuitively to understand if visualized, see figure 5.21. Figure 5.21 is a solution to figure 5.12, which completely avoids using ports to attach the Coordinator to the layer structure. Instead, we are having two layered structures, each of them highlighting a different aspect. Planes are meaningful on a “macro” as well as on a “micro” level of an architecture.

The letters U and M stand for *user plane* and *management plane*, respectively.

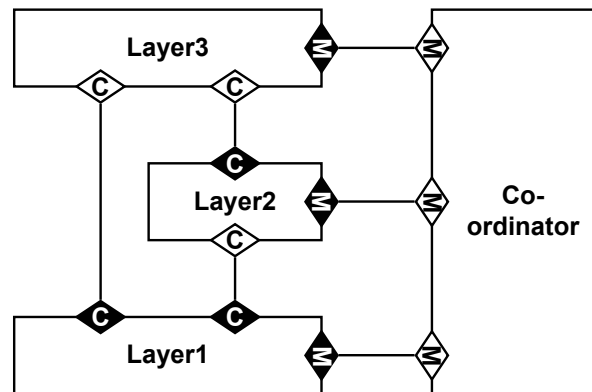


Figure 5.21: Planes exemplified

The user plane is given by the actor classes Layer3, Layer2 and Layer1, with Layer3 being a service user of Layer2 and Layer1, and with Layer2 being a service user of Layer1. The management plane is given by the Co-ordinator and the actor classes Layer1, Layer2 and Layer3, with the Co-ordinator being a service user of each of the other three actor classes.

The plane concept applies equally well to ports. It can be e.g. used to indicate that communication networks belong to different “spheres” of purpose. Thereby, a control network can be distinguished from a user network and a management network. Without a plane concept, the functional spheres would be less obvious.

In general, the plane concept lets a system architect indicate how different functional aspects of a system architecture are spread over the system and where they intersect. For example, in figure 5.21 the user plane and the management plane intersect at the layer actor classes; they all have to implement user and management functionality. Only the Co-ordinator actor class is hosting solely management related functions.

If we take the complex connector as a tool to stepwise break down functional aspects (as discussed at the end of the previous section) and see this technique in the light of a plane concept, complex connector refinement could be helpful in keeping plane aspects as much separated and thereby *orthogonal* as possible. Within the scope of this work this combined use of planes and complex connectors has not been fully explored and is left for further research. Yet we think that planes and complex connectors are also a very stimulating contribution to other research areas like aspect-oriented system development, see e.g. [AK03], and Architecture Description Languages (ADLs).

Notation

In the example in figure 5.21, we symbolized the plane a port, SPP or SUP belongs to by a small capital letter inside the interface symbol. This notation is based on an initial proposal by the author [HM01] but conflicts with the possibility to depict data interfaces by crossing the interface symbol. However, if we rethink what the plane concept really is, we can conclude that a plane defines a *namespace* for an interface. Only interfaces of the same kind and of the same namespace referring to the same protocol⁴ are allowed to be connected; the interface name is of no relevance.

Consequently, for ports and SPPs/SUPs we extend the notation for interface names by a preceding namespace name. Since namespaces can be organized hierarchically, the usual “dot” notation is used. Formally, a port/SPP/SUP can be now annotated by an address type, a namespace (hierarchy) and a name. Any of these parts is optional:

```
[<addrType> :] [<nameSpace>.*][<portName>]
```

An example for this is

```
IPAddr : ControlPlane.sender
```

which stands for an interface called “sender”, which is of type “IPAddr” and belongs to the namespace “ControlPlane”. This notation shall be used if data-oriented interfaces are used in the diagram. Alternatively, a modeling tool may associate a color code with a namespace and color interfaces to visually emphasise planes in a ROOM diagram. This makes models easier to read on e.g. a color monitor.

5.4.3 Architecture Modeling with Planes

Taking a closer look now at figure 5.22 reveals that it is a possible but concrete and precise architectural realization of the informal diagram in figure 5.20. We can identify two planes, a user and a control plane, each with two independent layers. For the sake of brevity, the access to the physical layer is not shown. The actor class at the very top provides a management SPP to a service user, and accesses the user and the control plane; the actor class fulfills the management function of plane coordination like SCF does in ISDN. In addition to that, each actor class of the user and the control plane provides a management SPP, which allows to access plane/layer specific resource and parameters functions (the other aspect of the management plane). With the upmost actor class as a shared layer resource included, we can count three layers in the user plane, three layers in the control plane, and two layers in the management plane.

⁴Remember that ROOM protocols depreciate to message schemata.

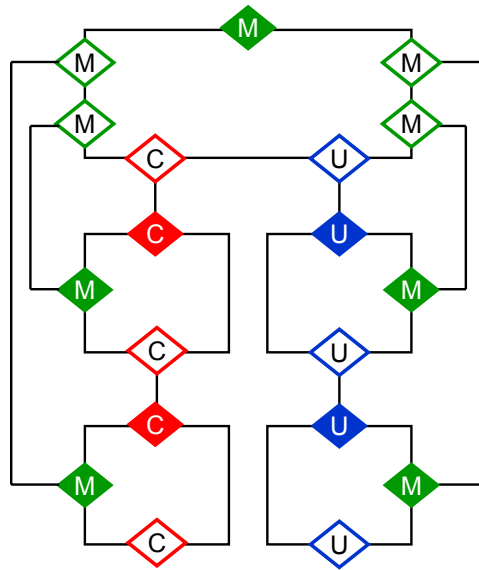


Figure 5.22: Precise architectural model based on ISDN, fulfilling figure 5.20

Another possible architectural solution is shown in figure 5.23. Here, there is still a coordinating actor class at the very top, but there are three interconnected planes: Actor classes of the management plane have a peer-to-peer client/server relationship to actor classes of the control plane, and actor classes of the control plane have an analogous relationship to user plane actor classes. This solution requires that all planes have the same number of layers. Given that the server actor classes have the capability to incarnate and destroy their clients, we end up with a highly dynamical architecture in which the management plane composes the control plane, which in turn composes the user plane. This architecture in fact is a lightweight realization of the Modular Communication Systems (MCS) reference framework as described in [Boe00].

We do not want to judge the solutions presented, but rather, demonstrate that a variety of architectural solutions can be described with the plane concept, which all fulfill figure 5.20. The preciseness achieved is a significant improvement over informal descriptions, and enables system architects to uniquely specify and communicate the functional organizational structure of planes, layers and communication networks of a communication system. The plane concept can be used to describe the architectural design of “traditional” solutions, like ISDN, as well as quite modern approaches such as the MCS framework. The examples also show that we do not need to introduce any further semantical constraints on ports/SPPs/SUPs other than discussed above. Layers of different planes might be interconnected via ports, yet we promote not to connect layers of one and the same plane

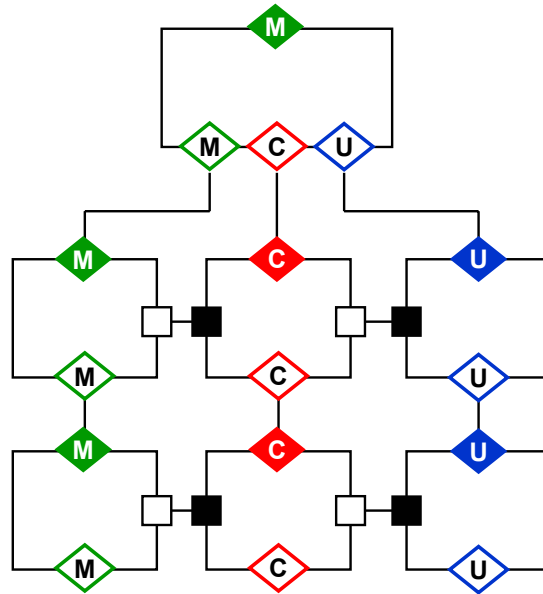


Figure 5.23: Precise architectural model based on the MCS framework, fulfilling figure 5.20

via ports.

The use of an explicit notation for planes puts us into a position to define and identify architectural patterns of the overall functional organization of a system. This is a novel aspect for the architectural design of communication systems and subject to further research.

5.5 Summary

In this chapter we discussed the notion of *layering* in a communication system. We began with a precise mathematical definition of layering. In communication systems, layering is a form of refinement or, more precisely, it bases on *communication refinement*. The communication relation between communicating parties – abstracted by a complex connector – can be refined by another, “lower” communication network (Equ. 5.5). Communication refinement retains the syntactic interface (Equ. 5.1, 5.3 and 5.4). The fundamental constraint put on the “lower” communication network is that its communicating parties must fulfill the *identity relation*. The identity relation demands that the communicating parties preserve the integrity of the information flow they process (Equ. 5.6). Herewith, communication refinement lets us rationalize why communication protocols are nested and why the refinement relation constructs an abstraction hierarchy.

We then took a closer look on the process of refining a complex connector. If the refinement is interpreted as a replacement relationship, we end up in a *node-centric* decomposition frame. If the refinement is interpreted as a nesting relationship, we end up in a *network-centric* decomposition frame. Both approaches adhere to the view of a communication system as an arrangement of communicating parties and complex connectors. Figure 5.24 summarizes both application cases of communication refinement.

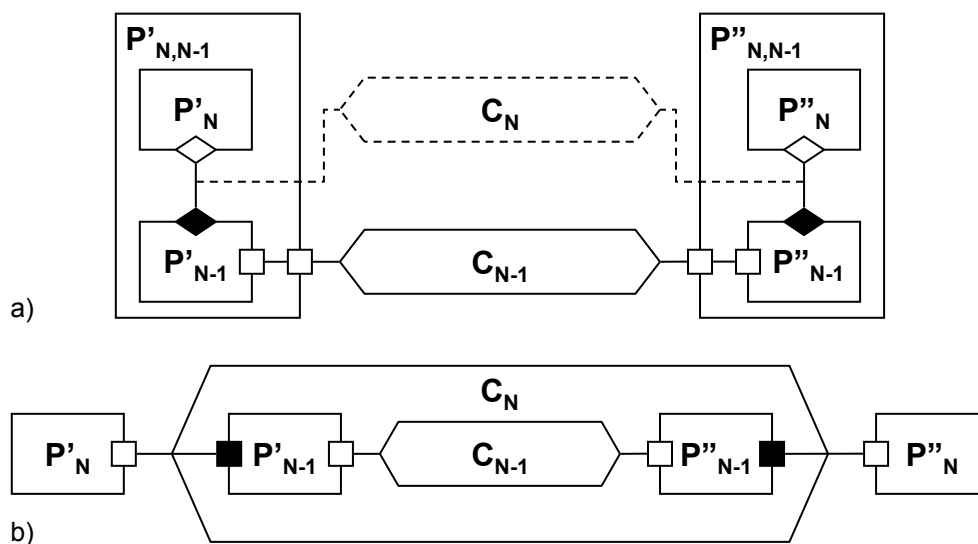


Figure 5.24: Communication refinement summarized: (a) node-centric, (b) network-centric

Next was an in-depth discussion about the “traditional” interpretation of layering and its realization in ROOM. “Traditional” layering is in total contrast to layer-

ing as communication refinement. Layering in the casual sense relates two entities in a *allowed-to-use* relation of service, one entity acting as a *service provider*, the other as a *service user*. That is all. This understanding of layering has nothing to do with abstraction.

The realization of this form of layering in ROOM was subject to a lot of criticism, mostly because of some confusing conventions in ROOM's terminology and notation. In a first step, we improved ROOM in that respect: we agreed to be explicit about layering (ROOM originally hides a lot of valuable details behind the construct of a layer connection), call the layer interface of the service provider *Service Provisioning Point* (SPP), the layer interface of the service user *Service Using Point* (SUP) and annotate both in ROOM diagrams. We introduced a symbol similar to ports. SPPs and SUPs are also connected via bindings.

These new conventions did not change the layering semantics of ROOM but helped elaborate that there is only a small but decisive difference between SPPs/SUPs and peer ports: SUPs are not allowed to be left unbound; an SUP cannot be connected to the behavior component of the composing actor class but must be exported or bound to a corresponding SPP. This simple rule ensures that service user and service provider are always independent concerning their compositional context and their lifetime.

This guarantee of contextual and lifetime independence is exactly what we want to have when we refer to *vertical communication*. So, by definition, we use SPPs and SUPs between communicating entities that are said to relate to each other in a vertical communication direction. For *horizontal communication* we use solely ports. This convention explains why we change ports to SPPs and SUPs if we resolve communication refinement in a node centric manner, see figure 5.24; the syntactic interface remains not impacted.

We then showed how communication refinement works in practice and exemplified it on relating the TCP user communication network and the TCP provider communication network, and on relating the UDP user communication network and the UDP provider communication network. When we considered distribution for vertical communication, exemplified on MGCP, we made an interesting and important observation: communication refinement cannot be only used to relate "pre-designed" communication networks but also to detail a communication relation into strata of functional aspects. Communication refinement turned out to be a much more powerful tool than for what we motivated it for.

Finally, we looked into the concept of organizing a system architecture in *planes*. Planes shape the structure of almost all modern networked communication systems; they are a means to organize a system into functional spheres. However, planes can be useful on any architectural level. On a high level the distinction in a *user plane*, *control plane*, and *management plane* is common. We introduced planes to ROOM as a *namespace* attribute extension to ports, SPPs and SUPs.

Chapter 6

Language and Implementation

Chapter 3, 4 and 5 introduced a lot of improvements and enhancements to ROOM. So far, the discussion of all these improvements and enhancements has been rather informal. Now, it is time to provide a precise specification of the language changes to ROOM. These changes led to a redesign of ROOM called ROOM++. Feature-wise, ROOM++ contains ROOM as a subset and is specifically adapted to the needs of modeling the system architecture of (tele)communication systems. The implementation of ROOM++ is called PyROOM++ since Python has been used as a programming and an action language for ROOM++.

In section 6.1 we elaborate the meta-model of the ROOM language. In section 6.2 all improvements and enhancements to ROOM are summarized before the meta-model of ROOM++ is explained. The meta-model is a formal description of all constituting language concepts of ROOM++ including their semantic dependencies. The meta-model is the basis for the implementation, PyROOM++, and is described in section 6.3. An additional section about an improvement to high-level behavior specifications, see section 6.4, rounds off this chapter about the ROOM++ language and its implementation. Section 6.5 closes with a summary.

6.1 The Design of the ROOM Language

For the design of our extended ROOM language, ROOM++, we have chosen to go for a four layer meta-data architecture and use the Unified Modeling Language (UML) as a meta-language. That is also why we present the design of the ROOM language as a meta-model in UML. Since a future goal of this work is to strive for integration with the forthcoming version 2.0 of the UML (possibly as a so-called *profile*), it has been a reasonable choice to base the language design on the same architectural framework and the same meta-language as the UML does. However, other approaches are not necessarily less meaningful.

The technique to structure a language in layers of meta-data has become especially popular with the design of the UML. As a matter of fact, the UML standard has been defined using UML itself as a meta-language. This design approach is called *meta-circular*, see e.g. [ASS96], and is in practice “broken” by a method called *bootstrapping*. To minimize bootstrapping efforts, one usually uses only a limited feature set of the language for the definition of its own meta-model. This meta-circular approach has been used to some extent by the authors of the ROOM book to explain the functioning of the ROOM Virtual Machine: The interpretation of a ROOM model is described in ROOM.

Before we present ROOM’s meta-model, we give a brief introduction to the four layer meta-data architecture as it is e.g. promoted by the UML standard [OMG01] and the Meta Object Facility (MOF) 1.4 standard [OMG02]. After that we explain, how we retrieved ROOM’s meta-model.

6.1.1 The Four Layer Meta-data Architecture

An overview of the four layer meta-data architecture is shown in figure 6.1. The layers are numbered from M0 up to M3 and are related in an *instanceOf* relationship. Loosely speaking, M(N-1) being an instance of M(N) means that M(N) is a specification of M(N-1).

- M3** M3 is the highest layer and describes a language, a so-called *meta-language*. The meta-language specifies basic language conceptions, which are allowed to be used on the next lower meta-layer. For example, the meta-language may specify a class concept, the meaning of inheritance, class attributes, methods, and an association concept in order to express an M2 specification in terms of classical OO-concepts. Such an example of a meta-language is MOF (Meta Object Facility), which was standardized by the OMG (Object Management Group) in 2002 [OMG02]. Very often, the UML is taken as a meta-language including OCL (Object Constraint Language) [OMG01,

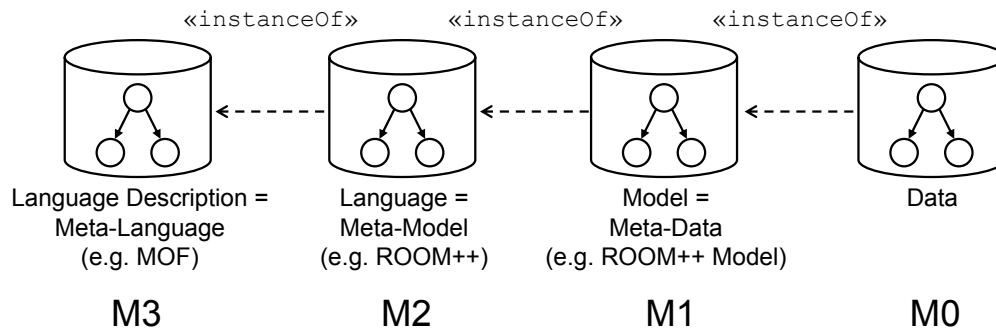


Figure 6.1: The four layer meta-data architecture

chap.6] as a means to express constraints in a model. In compiler construction, the EBNF (Extended Backus Naur Form) (see e.g. [Cro82, sec.2]) could be regarded as a meta-language.

M2 On M2, the actual language is defined with the help of M3 conceptions. On this layer, the (modeling) language ROOM and our extended version PyROOM++, respectively, are being specified. Here we find the specifications for the ActorClass concept, the Port concept, the ActorRef concept and so on. If MOF or UML is used as a meta-language, these concepts are defined as classes and set into relation via associations. It is also said that M2 describes the *meta-model*.

M1 A concrete *model* is specified on layer M1 using the conceptions defined on M2. All the ROOM(++) models we defined throughout this work belong to M1.

M0 At run-time, the M1 model is instantiated and exists as *data* on M0. For example, a simple model specified on M1 that consists of a single actor class with a port of replication factor three is represented on M0 as an actor class instance (simply called actor) and three ports.

In principle, one could have an infinite number of meta-data layers, which is sometimes useful to have. For our purposes, the four layer meta-data architecture is sufficient.

6.1.2 From a (Semi)Formal Specification towards a Meta-Model

There is no meta-model delivered with the ROOM book. Instead, a “formal” specification of the language is presented in an *abstract syntax*. Unfortunately, the abstract syntax is neither formally defined nor does it capture all semantical details of

the ROOM language. It is not possible to understand the ROOM language solely by reading the specification. That is why we rather prefer to classify the abstract syntax as a semi-formal specification of the ROOM language. The good news is that despite of its deficiency the abstract syntax for ROOM is sufficiently precise for a human reader (along with the material of the textbook) to allow a language designer to grasp all language conceptions and to reconstruct the meta-model of ROOM. The process of translating the formal specification into a meta-model can be even automated to a certain extend. Some manual intervention is needed to clean-up the meta-model, clarify some relationships, and decide especially on the type of the association to be used and on the multiplicity at the association ends. In some cases the meta-model benefits from a more compact representation. The advantage of a semi-automated translation process is that the meta-model becomes coherent with the formal language specification as presented in [SGW94] and is less colored by an individual's understanding and interpretation of the ROOM language. The rules applied for the translation process are described in the following.

The Abstract Syntax

The abstract syntax defines language concepts as tuples. For example the actor-interface concept [SGW94, p.187] is defined as a 3-tuple

$$\text{actor-interface} = \langle \text{peer-interface}, \text{sp-interface}, \text{impl-interface} \rangle$$

Elements of a tuple can be further refined by another tuple, i.e. the tuple element refers to another concept definition. Alternatively, a tuple element may stand for a set of mouldings of a concept, which let us interpret the set as a list. Sets are always indicated by the use of curly braces. For example, the peer-interface is defined by a list of port references:

$$\text{peer-interface} = \{ \text{port-ref}_1, \text{port-ref}_2, \dots \}$$

The referred concept definition is also indexed to indicate that different mouldings are distinguished:

$$\text{port-ref}_i = \langle \text{port-ref-name}_i, \text{replic-factor}_i, \text{protocol-class}_i, \text{conjug-ind}_i \rangle$$

Concepts that are not resolved further are sometimes restricted in their value, which introduces a simple type concept. For example

$$\text{conjug-ind}_i \in \{ \text{true}, \text{false} \}$$

This is basically the style how the whole ROOM language specification is noted down. Some few semantic constraints are given in a functional manner; many constraints, however, come in plain english and are spread over the ROOM book in the respective chapters introducing and explaining the language.

Translation Rules

The following rules help automate the transformation of the ROOM language specification into a meta-model. Here, UML is used as a meta-language for the meta-model.

1. Any concept definition is translated into a class, which we also refer to as a *concept class*.
2. If a tuple element of a concept definition refers to another concept definition, an association between the two concept classes is established. By default, the association is of type “composition”: the original concept class is *composed of* the referenced concept class.
3. If a tuple element of a concept definition refers to a list, an association between the concept class of the definition and the concept class referenced by the list elements is established. The association end closest to the implicitly referenced concept class is named according to the name of the tuple element. The multiplicity is “many” or “0 . . *” in UML terms.
4. If a tuple element is not resolved further, the tuple element is converted to an attribute of the concept class. The attribute retains the name of the tuple element. Note that this attribute in fact may be the pointer of an association end towards another concept class. That is to keep in mind for manual revision.
5. The following conventions for determining the type of an attribute apply:
 - If the tuple element is restricted on its value, the attribute is of type *enumeration*. If the value can be of “true” and “false” only, the attribute type is *boolean*.
 - Attribute names ending on “-name” are of type *string*.
 - Attribute names ending on “-factor” are of type *integer*

The use of compositions as the default for associations is reasoned in the design approach of languages in general, no matter of if we talk about programming or modeling languages. Formal languages typically aim to nest their structural concepts in such a way that the inner concept is only valid in a specific context of another concept. That is, an outer structural concept “owns” the inner structural concept. This sort of hierarchical nesting is captured best by a composition association. Other sorts of associations are less likely to appear. The composition association can be regarded as a first best guess.

As mentioned, the rules do not fully automate the translation process. While they produce a skeleton, which includes all key concepts, the relations and semantics need to be manually checked and revised.

6.1.3 ROOM Meta-Model

The outcome of the translation process is shown in figure 6.2. The figure shows the part of the ROOM meta-model that covers high-level structure modeling including layering; this corresponds to the material presented in chapter 6 and 7 of the ROOM book [SGW94]. Note that we excluded the aspect of high-level behavior modeling in our work.

The diagram does not show the constraints (so-called well formed rules), which apply. For example, ROOM demands unique class names; one cannot deduce this information from the class diagram unless explicitly stated.

ROOM's meta-model is shown here for comparison purposes with its improved version ROOM++. We will not provide further commentary on ROOM's meta-model since we give a very detailed presentation of ROOM++ including all well formed rules that shape the language. We would just like to draw the reader's attention to the following classes: ActorClass, ActorRef, PortRef and BContract will "survive" the re-design, partly with a different name in ROOM++. The other classes are either not needed or are replaced by more powerful concept classes.

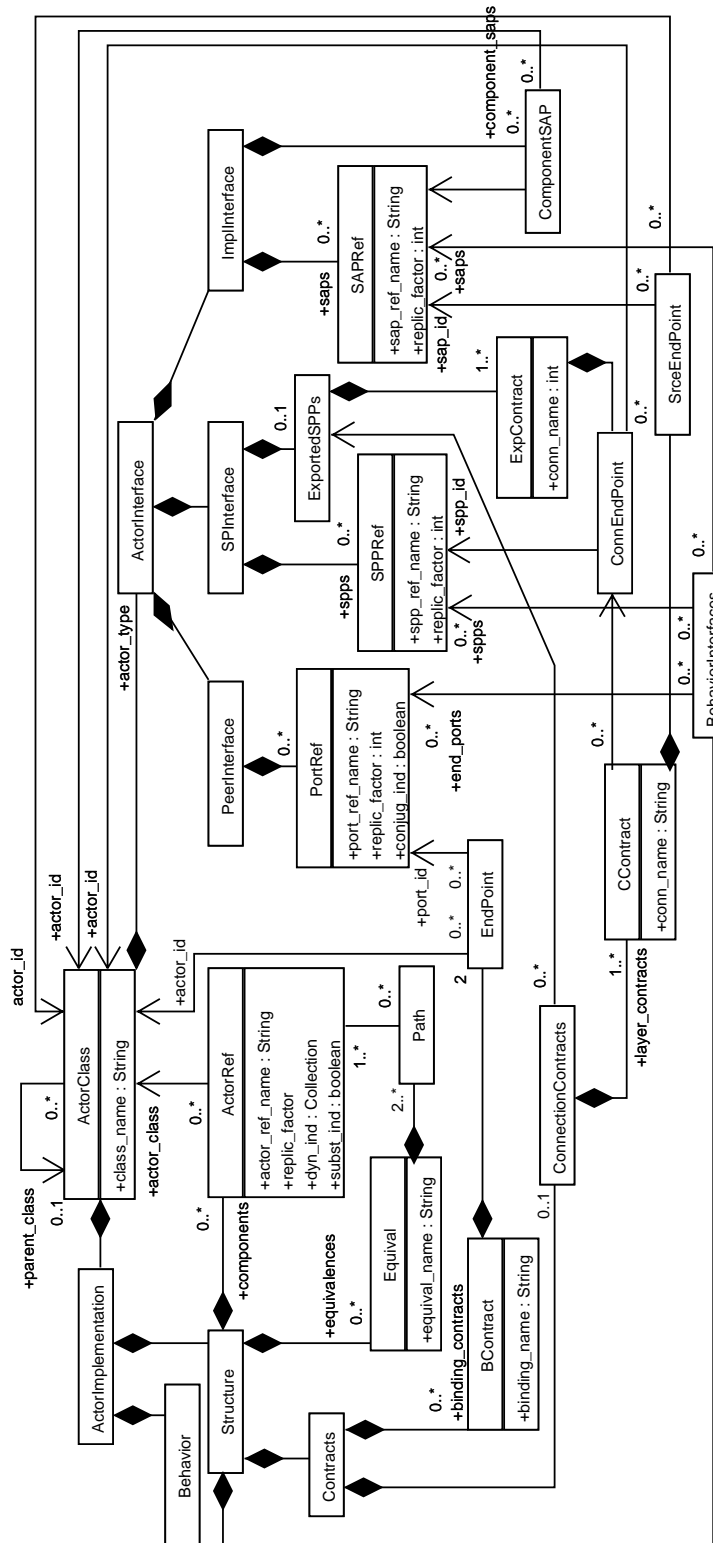


Figure 6.2: Meta-model of ROOM

6.2 The Design of ROOM++

Before we have a look at the meta-model of ROOM++, we summarize all the enhancements to ROOM we proposed in the previous chapters.

6.2.1 Summary of Enhancements to ROOM

Throughout the previous chapters, we extended and improved the ROOM language step by step. The following list summarizes the extensions and improvements that characterize ROOM++. The summary is not in chronological order as they have been evolved in this work.

- Unification of port/SPP/SUP concept
- Control and data ports
- Controlled Domain Models (CDM)
- Address types, address lists and address topologies
- Planes
- Relaxed port semantics
- Typed bindings
- Classification system

Subsequently, we go through all these bullets, provide a brief explanation of the ROOM extension/improvement, repeat the visual notation we agreed on and supply a short note on the run-time impacts. After this, we will see, how the extensions shape the meta-model of ROOM++.

Unification of Port/SPP/SUP Concept

In chapter 5 we thoroughly investigated ROOM's realization of layering that is materialized by the concept of a SPP and a SUP (or SPP and SAP according to ROOM's original terminology). We elaborated that the port concept and the SPP/SUP concept are so close that they can be harmonized and unified. There is only a single rule that requires SUPs not to be left unbound.

In ROOM++ the concept unification looks like follows: ROOM++ only has a *port* concept with the known attributes *name*, *replication*, and *conjugation*. An additional flag named *sipKind* indicates whether the port may have slightly different binding semantics. SIP stands for *Service Interaction Point*. If a port is of kind SIP

and conjugation is set to false, we call the port *Service Provisioning Point* (SPP); if the port is of kind SIP and conjugation is set to true, we call the port *Service Using Point* (SUP). Semantically, only ports of the same kind can be connected via a binding (rule 1) and a port of kind SIP and conjugation set to true (actually, a SUP) must have an external binding (rule 2).

In our ROOM++ diagrams, we annotate ports of kind SIP by a slightly different symbol: the port symbol is rotated by 90 degrees. Not conjugated ports are filled in black, conjugated ports are filled in white, see figure 6.3. Replication greater than one is indicated by a “shadow” symbol; the exact replication factor is not included in the graphical representation. Note also that we do not favor extra symbols for end ports, reference ports and relay ports in our diagrams like ROOM does. These are only naming conventions depending on where the port symbol appears, on the border of an actor class symbol or actor reference symbol, and whether it has an internal binding or not. While we believe the terminology to be meaningful, we regard the introduction of extra symbols as superfluous and visual clutter. Semantically, there is no different between ROOM and ROOM++ in this respect.

The unification is uncomplicated and elegant since it imposes only the two above mentioned simple rules on the modeling level but has no impacts on model interpretation at run-time. Compare this to ROOM’s meta-model. At run-time, messages are only sent between port instances.

Control and Data Port

In chapter 3 we introduced the option to mark a ROOM port either as a *control port* or as a *data port*. We call this the actor’s *style*. Later, in chapter 5, we applied the same semantics to SPPs and SUPs. We extended ROOM’s notation in the following way: for control ports (which includes SPPs and SUPs) we attach a small arrow directly to the interface but inside the actor class; for data interfaces (again including SPPs and SUPs) we simply “cross out” the interface symbol. An interface that is neither marked as control-oriented nor data-oriented is said to be unspecified with regard to its style. As a matter of fact, we added a style attribute to the concept of ports. Figure 6.3 summarizes the notation for unspecified, control and data ports of casual or SIP kind.

Note that for model execution the distinction in control and data ports is of no relevance.

The Controlled Domain Model

For control ports we promoted an architectural style of grey box specification, see chapter 3. The notation we used to publish grey-box information (the Controlled

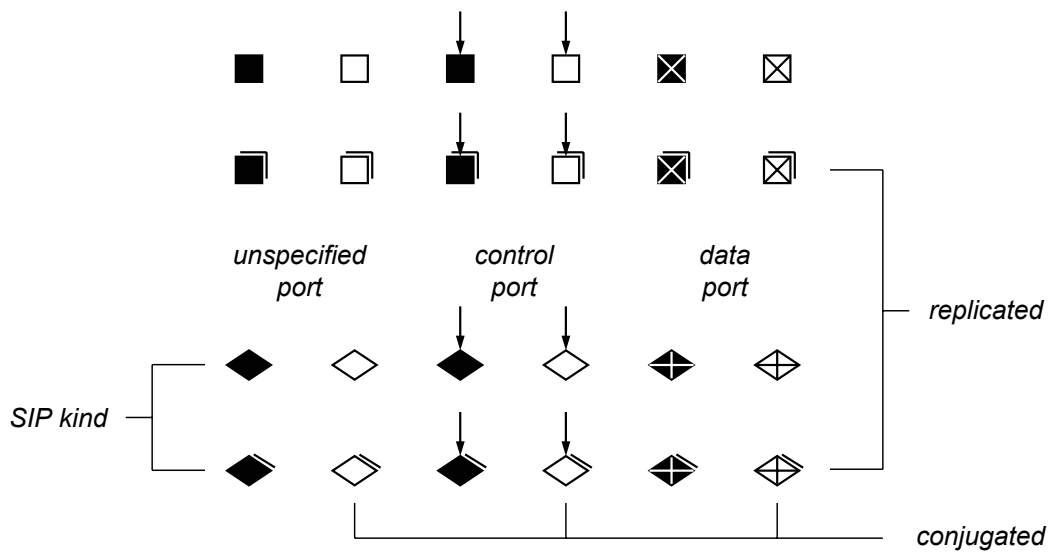


Figure 6.3: Notation for unspecified, control and data ports, SPPs, and SUPs

Domain Model, CDM) was informal and not manifested by a binding visual syntax. For simple cases, when the CDM is contained in a single actor reference or data class, we proposed to draw the respective element on the border of the actor class symbol to highlight the grey box nature of information, see figure 6.4. Admittedly, this is not the best visual notation. On a computer aided graphical modeling front-end better alternatives are thinkable. For example, if the user moves the mouse pointer upon the control port symbol, the tool may pop up a diagram of the CDM, which usually is a subset of the actor class' inner structure.

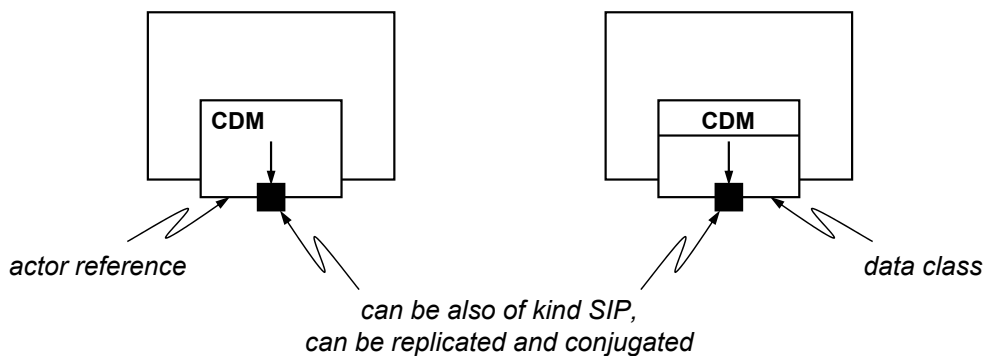


Figure 6.4: Notational proposal for a simple CDM

The CDM is purely architectural information and of relevance only on the modeling level. Model execution is not impacted by the absence of a pointer to the CDM for a control port.

Address Type, List and Topology

Means to enhance ROOM by the capability to model address spaces were introduced in chapter 4. There, we extended the port concept by a typing concept (an *address type*) and the option to specify a list of concrete *addresses* that are assigned to instances of the port at run-time. In addition, a reference to the realization of the *address topology* may be given.

The address type is modeled by a data class that usually resides outside the context of the actor class specification. In the notation, the address type precedes the (optional) port name separated by a colon. The (optional) list of addresses is enclosed in curly braces, see figure 6.5. The (optional) reference to the data class or actor reference implementing the *address topology* is given by a dashed line connecting the port to the respective element.

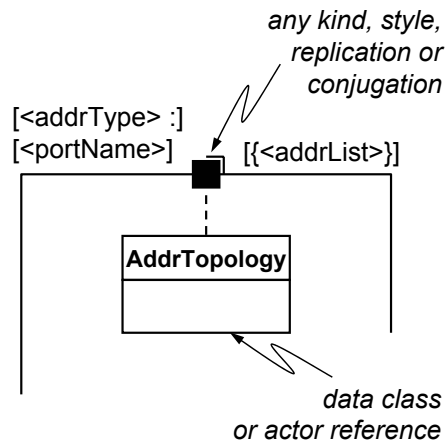


Figure 6.5: Notation for address spaces

Addressing has an impact on the run-time level and on the use of ports in behavior component specification. In ROOM, a port is incarnated as a set of port objects at run-time, each set having as many members as specified by the replication factor, and can be addressed by an index only. For example,

```
port[2]
```

refers to the third instance (indexing starts with zero) of a port. In ROOM++ we may associate an address type with the port, say an IP address, and write instead

```
port['137.226.168.58']
```

which refers to the port instance associated with the given address.

Planes

In chapter 5 we introduced the *plane* concept, which is a powerful tool and easy to install in ROOM++. An additional port attribute is reserved to optionally specify a plane. Only ports of the same plane are allowed to be connected. In the diagrams, planes are annotated much like a namespace attribute preceding the port name, see figure 6.6. A computerized tool may offer additional visualization effects.

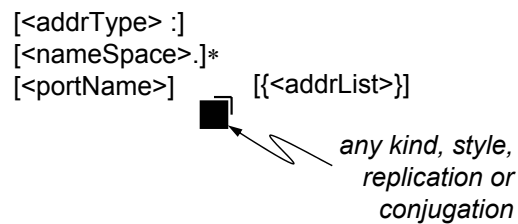


Figure 6.6: Notation for planes

Planes do not play a role at model execution time.

Relaxed Port Semantics

In contrast to ROOM we relaxed some port semantics. Foremost, we would like to mention the possibility to have *conjugation symmetry* for ports connected via a binding that are not of SIP kind. There is no need to demand that one port of a (typed) binding relationship must have conjugation set to false and the other to true. Conjugation is only for the comfort of not specifying a protocol twice, one version having the set of incoming and outgoing messages swapped. Still, it remains a matter of the run-time machine to verify if a message is allowed to pass a port according to the protocol specification or not. Conjugation symmetry is a non-critical relaxation to ROOM. An example is shown in figure 6.7.

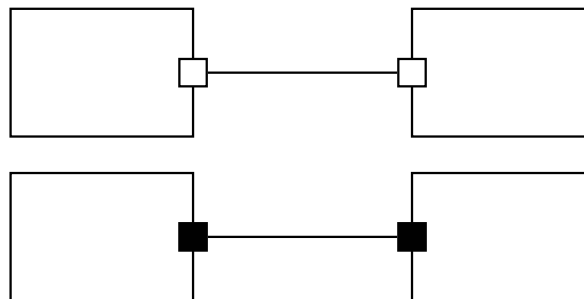


Figure 6.7: Conjugation symmetry in ROOM++

Another relaxation to ROOM is that we do not have *internal end ports* (see [SGW94, p.169]). We apply the simple rule that all external end ports of an actor class and all reference ports of its contained actor references, which are left unbound, are implicitly connected to the actor class' behavior component. Again, this avoids some visual clutter in ROOM++ diagrams and slightly changes message passing semantics in ROOM++. To emulate internal end ports, the behavior component must explicitly ignore messages received from other “open” ports.

Typed Binding

The complex connector and its realization in form of a *typed binding* were introduced in chapter 4. In ROOM++, the typed binding is modeled as a specialization of the binding concept in order to show (a) how ROOM++ evolves out of ROOM and (b) how typed bindings in ROOM++ make use of traditional ROOM bindings. In ROOM++, the typed binding is much like a binding but with the possibility to have more than two endpoints for n-ary complex connectors. The type is given by a pointer to an actor class and a set of binary (ideal) bindings specifying, which ports of the actor class belong to which endpoint of the typed binding.

For typed bindings the notation is shown in figure 6.8. The actor class, which specifies the behavior and type of the typed binding is written inside the arrow-like symbol. An optional binding name may be attached outside. The bindings connecting the type (actor class) with the endpoints of the typed binding are not shown in the graphical representation.

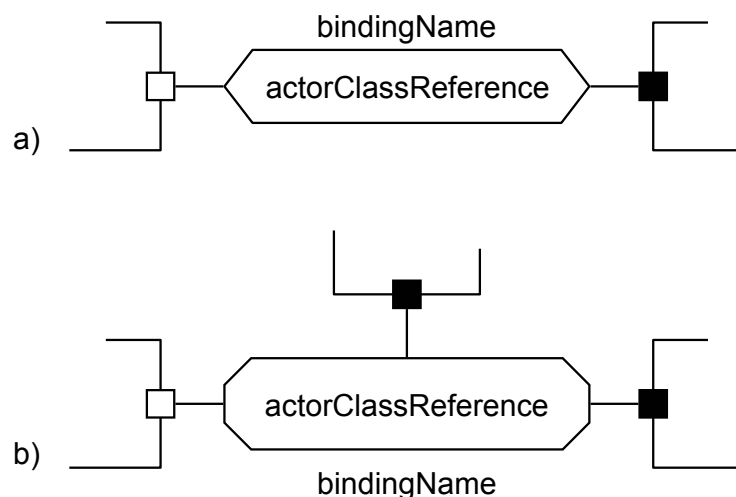


Figure 6.8: Notation for typed binding: (a) binary, (b) n-ary

The typed binding has run-time impacts. Instead of just establishing communi-

cation paths inside a ROOM run-time representation, in ROOM++ an incarnation of the typed binding's type is put in between incarnations of the communicating parties.

Classification System

In chapter 4 we started to use special symbols for specific kinds of actor classes. These symbols marked valuable architecture information and helped grasping the functional purpose of distinct actor classes quite easily. This feature calls for means to classify actor classes in categories if needs be. We will therefore introduce a category attribute in ROOM++ for actor classes. It is up to a modeling tool to associate a suitable visual representation to a categorized actor class and to an actor references or a typed binding that refers to a categorized actor class.

In this work we used an actor class symbol with rounded corners in order to indicate that this actor class may be also replaced by an n-ary typed binding having this actor class as a type. The reason was to have a methodological clean relation to the mathematical formalism used to reason our approach. Furthermore, we introduced a symbol for demultiplexers, see figure 6.9.

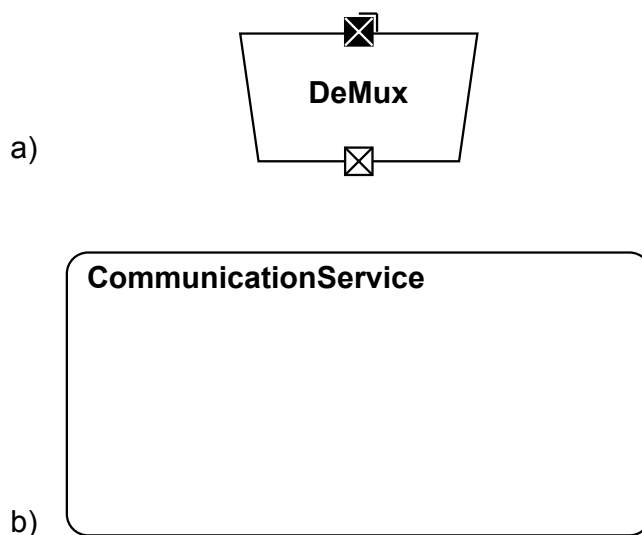


Figure 6.9: Notation for a) demultiplexer and b) communication service

At run-time, the categories of actor classes have no impact on the execution model.

6.2.2 ROOM++ Meta-Model

The meta-model of ROOM++ is shown in figure 6.10; the well formed rules that complement the meta-model are listed below. Note that for model explanation the following conventions apply: All ROOM++ core concepts are modeled as classes; their names are written with a leading capital letter, like ActorClass. Attributes as well as associations (associations are some sort of special attributes) are written in small letters, like name. References from one core concept class towards another concept class are given by the name of the corresponding association end, e.g. ActorClass refers to a set of ActorRefs via *components*; we then say that an ActorClass has zero or more components. If there is no role name provided for an association end, we refer to it using the name of the addressed concept class but write it with a leading small letter. For example, an ActorRef has one and exactly one actorClass (besides having a container). With these rules, model navigation should be unambiguous.

ActorClass

An ActorClass has some defined behavior and belongs to a model. In addition, it may have a name and may consist of a set of interfaces, a set of components and a set of contracts. The ActorClass can inherit these optional properties also from a superClass. In extension to ROOM, the ActorClass may be the type specifier for a number of typedBindings.

WELL FORMED RULE 1 (**validSuperClass**)

Tail and head of superClass must be different.

Whenever we refer to attributes and associations of ActorClass in the following, note that we assume that all attributes and associations of the superClass (and of the superClass' superClass, and so on) are implicitly addressed as well. This assumption makes the formulation of well formed rules less complicated.

Inheritance in ROOM++ is simpler than in ROOM. In ROOM++, values assigned to attributes of a subclass instance overwrite values taken over from the superclass instance. Association values of a subclass instance extend the association values of the superclass instance. In ROOM it is possible to exclude association values taken over from the superclass instance. The pros and cons of ROOM's inheritance mechanism are discussed in the ROOM book [SGW94, Chap.9].

Port

A Port belongs to an actorClass and optionally has a name. The replication factor is by default set to one, and the conjugation flag is by default set to false. The

`sipKind` attribute determines, whether a `Port` instance is a casual port (the default) or a *Service Interface Point* (SIP). If it is a SIP and `conjugation` is false, we call the port *Service Provisioning Point* (SPP); if `conjugation` is true, we call the port *Service Using Point* (SUP). This way, we harmonize the port and the SPP/SUP concept. The `style` attribute is either of value `control-oriented`, `data-oriented`, or `unspecified` (default). The `plane` attribute defines the namespace for the port.

Each `Port` must be associated with a protocol; it may be bound to bindings and refer to a `cdm` (Controlled Domain Model) and an `addrTopology`. To model address spaces, the `Port` can be associated to an `addrType` and to a list of addresses, with each address in the list being of type `addrType`.

WELL FORMED RULE 2 (**validBindings**)

If number of bindings is one, bindings must refer to a `Binding`. If number of bindings is two, bindings must refer to a `Binding` and a `TypedBinding`.

WELL FORMED RULE 3 (**validKind**)

If `sipKind` is true and `conjugation` true, then the `Port` must have bindings. This rule ensures that SUPs are not left unbound.

WELL FORMED RULE 4 (**validStyle**)

If kind is “SIP” and `conjugation` true, then `style` must not be of value *control*. This rule complies to our convention that control can be not assigned to a SPP.

WELL FORMED RULE 5 (**validCDM**)

- If `style` is not `control-oriented`, there can be no `cdm`.
- Subsequently, the `actorClass` owning the `Port` is called *owner*; and the other `Port` listed in the endpoints of the `Binding` the original `Port` refers to is called *other endpoint*: If the `actorClass` of the *other endpoint* is also referenced by one of the components of the *owner*, then there can be no `cdm` associated to the `Port`. In short, this rule says that relay ports/SPPs/SUPs cannot have a CDM.

WELL FORMED RULE 6 (**validAddrTopology**)

If there is no `addrType` given, there can be no `addrTopology`.

ProtocolClass

Each `ProtocolClass` may have a name and a list of names for `inMessages` and a list of names for `outMessages`. These attributes may also be inherited from a `superClass`.

WELL FORMED RULE 7 (**validSuperClass**)

Tail and head of `superClass` must be different.

ActorRef

A ActorRef has an optional name, a default replication factor of one, is of kind *fixed* (default), *optional* or *imported*, and attribute `substitutable` is set to false by default. The ActorRef is owned by a container and points to an actorClass.

WELL FORMED RULE 8 (**validActorClass**)

The associated actorClass must be different from container.

Binding

A Binding may have a name. It is owned either by an actorClass or by a typedBinding and refers to some endpoints.

WELL FORMED RULE 9 (**validOwner**)

A Binding must be owned by either an actorClass or a typedBinding.

WELL FORMED RULE 10 (**numberOfEndpoints**)

The number of endpoints must be exactly two.

WELL FORMED RULE 11 (**validKindAndNamespace**)

Both endpoints must be of the same sipKind and of the same plane.

WELL FORMED RULE 12 (**validEndpoints**)

- No two endpoints are identical.
- If owned by an actorClass (called owner subsequently), each Port of endpoints must either be among the owner's interfaces or must be among the interfaces of the actorClass of an ActorRef that is among the owner's components.
- If owned by a typedBinding, one Port of endpoints must be among the interfaces of the typedBinding's type; the other Port must be either among the interfaces of the typedBinding's actorClass or among the interfaces of the actorClass of an ActorRef that is among the components of the typedBinding's actorClass.

Note that we do not demand that the one of the endpoints of a Binding must have `conjugation` set to false and the other to true. Nor do we formulate any demands on the consistency of protocols for endpoints. Valid messages are checked at run-time according to the associated protocol of endpoints disregard of the *conjugation* flag.

TypedBinding

A TypedBinding inherits all attributes and associations from Binding. A TypedBinding must have a type and contains two or more bindings.

WELL FORMED RULE 13 (numberOfBindings)

The number of bindings a TypedBinding owns must match the number of endpoints the TypedBinding refers to.

WELL FORMED RULE 14 (validType)

The actorClass and the type of a TypedBinding must be different.

WELL FORMED RULE 15 (validEndpoints)

- No two endpoints are identical.
- The actorClass owning the TypedBinding is called owner subsequently. The actorClass of each Port the endpoints of a TypedBinding refer to is identical to an actorClass of the owner's components. (Note, this rule is more verbosely described in "Remarks" on page 126, chapter 4)

GreyBoxInfo

The GreyBoxInfo refers to a number of components and dataClasses as the constituting elements of either a controlPort or an addressPort.

WELL FORMED RULE 16 (validComponents)

The actorClass of a controlPort /addressPort of a GreyBoxInfo is the same as the container of all components.

WELL FORMED RULE 17 (validDataClasses)

The actorClass of the controlPort /addressPort of a GreyBoxInfo is the same as the actorClass of the «PyClassType» that is used by «PyTypeType» of dataClasses.

«PyClassType»

Our implementation of ROOM++ is closely integrated with Python to take full advantage of Python's object-oriented features. At two places in the meta-model we refer to «PyClassType», which denotes not a Python class but a Python type that – if instantiated – becomes a Python class. Herewith, we allow the modeler to specify Python classes on meta-data layer M1 and refer to such a class as (a) a addrType of Port or as (b) a behavior specification for an ActorClass. In case (a), instances of «PyClassType» make up the list of addresses the Port may refer to. In case (b), it is the run-time environment, the ROOM++ VM, that instantiates the behavior that is specified as a Python class on M1. This way, we delegate all behavioral issues to Python and use ROOM++ as the structural framework.

«PyTypeType»

«PyTypeType» is another element in the model to express the wiring to Python. «PyTypeType» denotes a generic type in Python generalizing all Python types available. This includes, for example, a class type (which we referred to as «PyClassType»), a list type, a string type, an integer type, a method type and so on and so forth. In the meta-model, we use «PyTypeType» to indicate that on M1 layer any interna of a Python class specification (expressed by “PyClassType *uses* PyTypeType”) can be subject of reference by the GreyBoxInfo instance; for naming compatibility with ROOM, these references are called `dataClasses`. The GreyBoxInfo can expose not only structural components but also details of the behavior Python class specification.

Practical Considerations

For convenience purposes, to ease a human modeler the specification of a ROOM++ model, the name attribute of ActorClass, Port, ActorRef, (Typed)Binding and of ProtocolClass may be used as a unique identifier for the respective element instance. As a consequence, the modeler can use names instead of M2 instance IDs to specify association values. However, this technique is impractical for specifying the endpoints of a (typed) binding. Since ports are defined in the context of actor classes, an endpoint may be alternatively specified by the name of either an actor class or an actor reference and the name of a port.

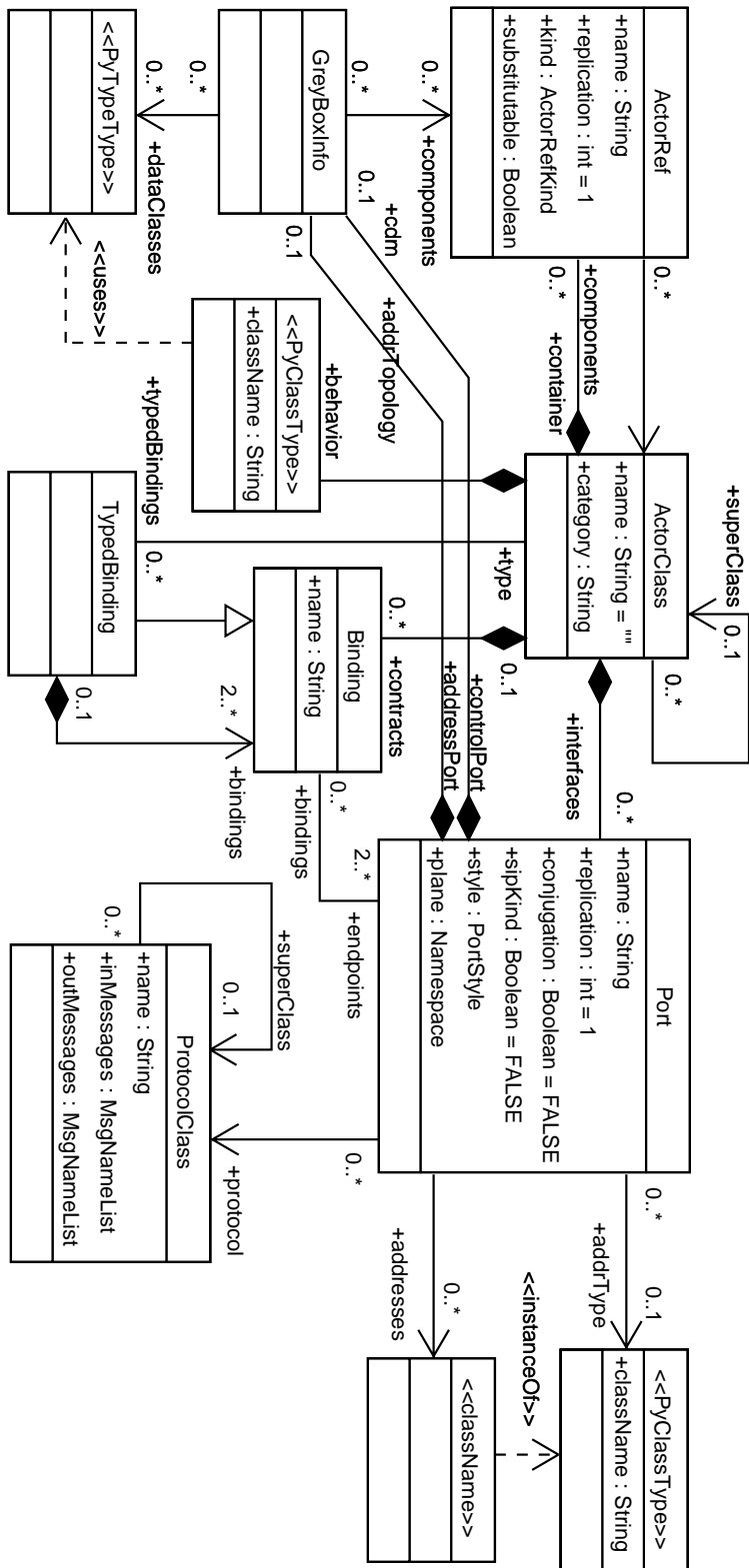


Figure 6.10: Meta-model of (Py)ROOM++

6.3 The Implementation: PyROOM++

In this work we focused on architecture as a matter of structure and organization of a system and used ROOM as a base language to describe structure and organization in terms of actor classes, actor references, ports and bindings. We presumed that behavior, which is encapsulated in behavior components and/or data classes, is just specified somehow. We did not make any further assumptions about the *how* of behavioral specifications. We just said that state machines are one convenient way to do so. UML's activity diagrams might be another option, SDL's process diagrams a further one. Key only is, that the behavioral specification can digest and emit messages.

For the implementation of ROOM++ we decided that an actor class' behavior has to be "specified" in form of a Python program; this includes also the realization of data classes. Python is an object-oriented, interpreted, dynamically typed programming language, which is ideal for scripting and rapid application development and prototyping. The language is very well maintained, comes with an extensive standard library, and has a large and growing user base. Using Python as an embedded language for ROOM++ frees us from implementing a high-level behavior language and provides the modeler with an easy to use, "fits your brain" detail level action language. What makes Python especially attractive as an action language is (a) that it is dynamically typed and (b) its interpretive nature. Dynamic typing leads to very readable programs, which are almost on the level of pseudo code. This comes very close to the thinking of system architects, who want to express the general functioning but do not mind all details. Program interpretation, in addition, allows the system architect to immediately test and try out an actor class' behavior independent of its ROOM++ context. If wanted, one may provide state machines or any other suitable high-level paradigm for behavior specifications as a Python library. Since Python can be easily coupled with C, C++ or Java programs, it is also possible to use these programming languages for behavior "specifications".

ROOM++ itself is implemented in Python, so there is a tight integration of ROOM++ as a framework and Python as a programming and action language. Unfortunately, none of the existing ROOM tools was open to languages extensions, nor was the source code available. What we needed for our research was a clear, simple and extensible implementation of the ROOM language and full control over model specification interpretation and model execution. The only solution was to write an own implementation of ROOM and extend it for an uplift to ROOM++.

For application prototyping, Python proved to be useful and efficient. The design of PyROOM++ was first tested on a prototypical re-implementation of ROOM's virtual machine. This first prototype implemented almost 75 % of ROOM's

language features (no layering, no inheritance) and fitted into roughly 1000 lines of Python code. For ROOM++ the code base has grown up to 2500 lines of code, which is still relatively compact for such a complex virtual machine. Consider that Python programs are said to be 3–5 times shorter than the equivalent Java program.¹

6.3.1 Features and Accepted Shortcomings

Supported ROOM Features

PyROOM++ implements all features of ROOM and the above mentioned enhancements. To be explicit, PyROOM++ supports the following ROOM features:

- inheritance (actor classes, protocol classes)
- actor and port replication
- actor decomposition
- optionality (incarnation and destruction of an actor at run-time)
- multiple containment (import and deport of an existing actor in another context)
- asynchronous and synchronous communication (including timeouts)
- timers
- message priorities
- priority scheduler
- observability and controllability

Accepted Shortcomings

The implementation of PyROOM++ has three shortcomings:

- Most significantly, PyROOM++ comes without a Graphical User Interface (GUI). The most severe consequence is that models cannot be specified in a graphical manner on the screen. They have to be typed in as a series of Python statements creating and linking instances of the ROOM++ language

¹See e.g. <http://www.python.org/doc/essays/comparisons.html> (2003-03-08).

concepts. These are the statements that a GUI would perform in the background to internally set-up a model representation. In so far, PyROOM++ is prepared to be extended by a much more comfortable user interface. Such a minimalistic user interface is justified for a research prototype.

- Another shortcoming we already mentioned is that behavioral specifications do not follow a formal approach, they are rather Python programs. Even though we present this as a shortcoming, it can also be seen as an advantage: A small Python program for an actor class is often much more rapidly written and tested than a formal specification is. Since we regard behavior of our architecture models more as examples of use case scenarios rather than implementation sketches, the use of Python supports our vision of *rapid model prototyping* quite well. The intent is that system architects should be capable to experiment with alternative solutions on an architectural level, discuss and asses them as easily and quickly as possible. For that purpose, formal languages are more of a handicap and a restriction than a help. The motto *architecture scripting* indicates a different attitude. For some strange reason, system architecture development is often presented as a very serious business requiring whole product cycles to evolve the system and its architecture. In our vision, system architecture modeling and development is still a serious and very important business, but the architects are equipped with tools to “play” with the architecture, thereby having a much shorter feedback cycle. As a result, the architect can much more cheaply evolve and direct architecture development.
- Python is not suited for real-time programming. PyROOM++ runs each actor class incarnation in its own thread of control, so execution is concurrent, but e.g. Python’s automated garbage collection mechanism is not predictable and may conflict with hard real-time requirements. This statement does not mean that PyROOM++ cannot be used for modeling the architecture of a real-time system (quite the opposite is true) but its capability to correctly simulate strict real-time scenarios is limited. One improvement in this respect would be to extend the PyROOM++ VM by an event discrete simulator. This would make PyROOM++ useful even for performance and load analysis.

Intentionally not Implemented

The language feature to import an already incarnated actor class somewhere else in a ROOM model has been called *multiple containment*. A supportive concept for multiple containment is the notion of an *equivalence*. An equivalence specifies (a) the location of the actor that will be imported somewhere else, and (b)

the locations of the actor references that import this specific actor. Locations are described by a path downwards in the containment hierarchy of actor classes and actor references. The equivalence is owned by the actor class with the shortest paths to all locations.

The purpose of equivalences is questionable, because they overspecify a model. Multiple containment works fine without equivalences. An import actor reference unambiguously points to the actor class of which instances can be plugged in. There is no point in specifying this information a second time with equivalences. Consequently, we do not use equivalences and regard this language feature as an peculiarity (if not to say an oddity) of ROOM with no further implication on the language as such. It comes to no surprise that in later versions of the ROOM tool, see e.g. Rational RoseRT [Rat00], there is no equivalence concept anymore.

6.3.2 Implementation of Four Layer Meta-data Architecture

An interesting question is how a four layer meta-data architecture can be implemented using an “ordinary” programming language. We, for instance, implemented ROOM++ in an object-oriented, dynamically typed programming language called Python. In Python, like in many other programming languages, only two layers are under control of the programmer. The language itself, here Python, is given and corresponds to M2; there is not much to change.² So, what is left is the program level and the execution level: Python programs correspond to M1, their run-time representation to M0. The question is: How can a four layer meta-architecture be implemented using a programming language, which puts only two layers, namely M1 and M0, to the programmers disposal? One solution to this problem is to work with shifted cascades of M2-M0 layers, see figure 6.11.

In the upper cascade, we regard Python as the meta-language to define a meta-model of ROOM++. That means, concepts like ActorClass, Binding etc. are specified as Python classes, and a concrete ROOM++ model is an instance thereof. Since Python lacks some of the MOF/UML facilities like e.g. associations we may want to use in the meta-model specification, one could (a) either extend Python using its meta-programming facilities, or (b) provide a library with classes that offer associations and the like as helper functions, or (c) add another shifted M2-M0 cascade that implements e.g. MOF using Python. In practice, one may use ingredients of all three possibilities.

In the lower cascade, Python is used to specify execution concepts of the ROOM++ language, such as actors, ports etc. and to define the *instanceOf* relation of execution and ROOM++ concepts. M1 in the lower cascade *refers to* M1

²In case of Python, meta-programming is supported, which enables the programmer to change language semantics.

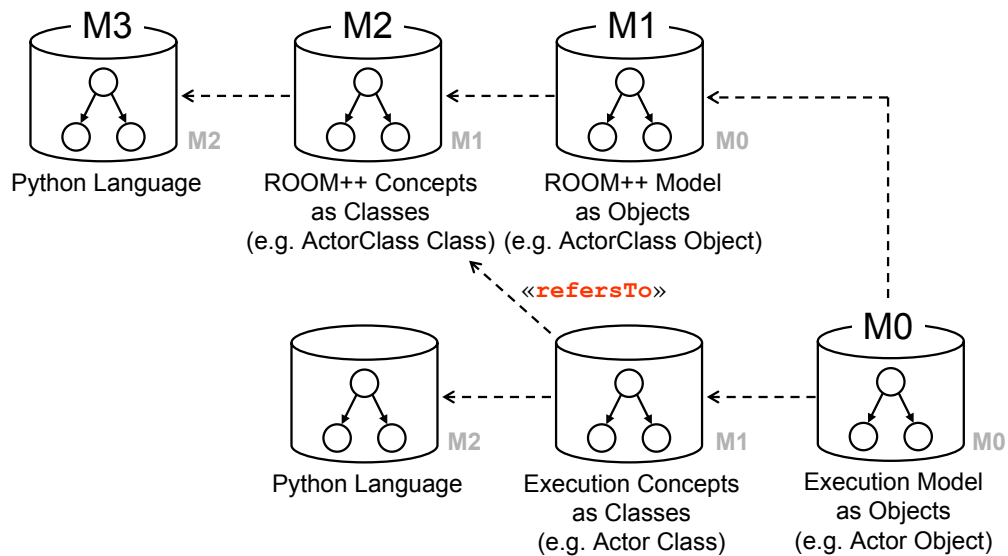


Figure 6.11: Implementation of four layer meta-data architecture via M2-M0 cascades

of the upper cascade in the sense that the lower M1 is the interpreter of the upper M1. *Refers to* is basically the definition of the *instanceOf* relation between the upper M0 and the lower M0 and is functionally equivalent to the *FrameService* of the ROOM Virtual Machine (VM).

Altogether, the upper cascade represents layer M3, M2 and M1 of the four layer meta-data architecture; the lower cascade contributes only M0. The *instanceOf* relations between M3-M2 and between M2-M1 are given by the use of Python. The *instanceOf* relation between M1-M0 needs to be specified explicitly in the lower cascade and is part of the ROOM(++) virtual machine. Further functionality of the VM, like the communications service and the processing service operate on M0, the data layer only. At run-time, the execution concepts are instantiated and relate to objects in the ROOM Model, M0 in the upper cascade.

6.3.3 Description of Basic Functioning

An overview of the functioning of PyROOM++ is shown in figure 6.12. The specification of a ROOM++ model distinguishes a **Structure** part and a **Behavior** part, the former referring to the later. In practice, both parts usually coexist in a single Python file. **Structure** is a sequence of Python statements describing a ROOM++ model, **Behavior** is the complementing behavior description for the actor classes. The **Structure** is processed by an engine that implements the ROOM++ language and verifies if **Structure** is a valid model specification. An internal ROOM++

model representation (remember, M1 is an instance of M2) is the outcome of the first processing step. This internal model of the structure specification is taken by the ROOM++ Virtual Machine, interpreted and leads to a run-time representation of the model specification (M0). Actor classes are incarnated together with their behavioral specifications.

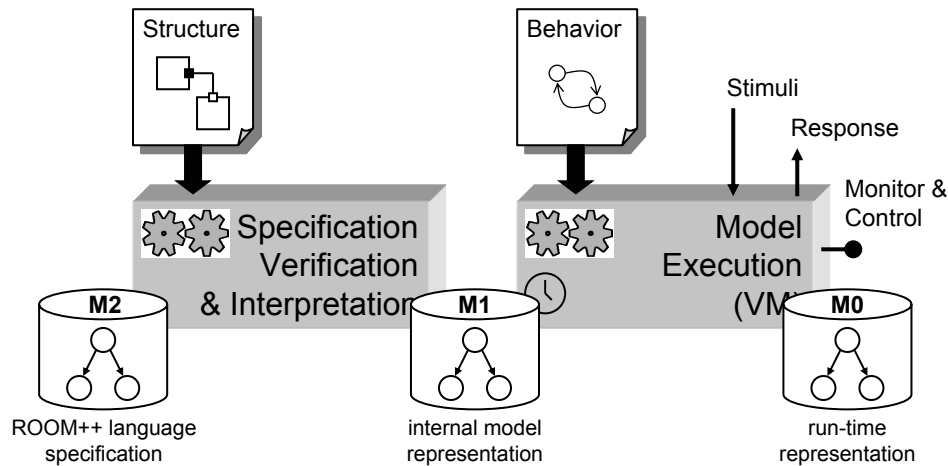
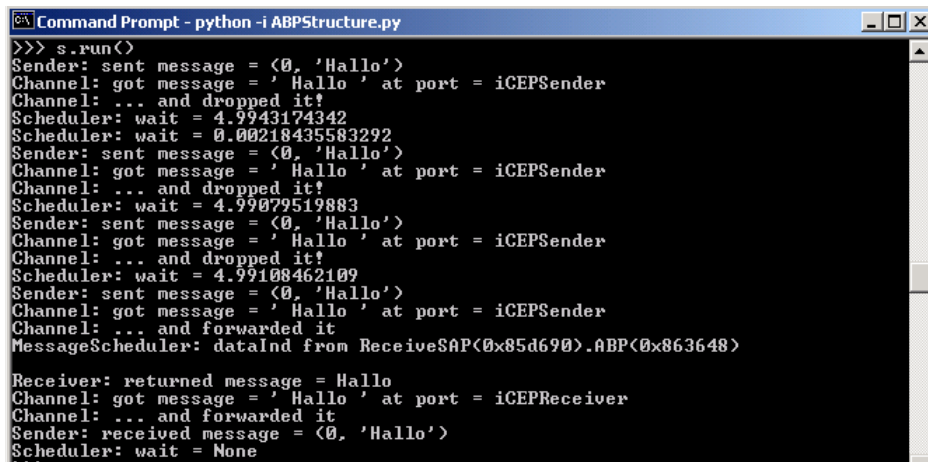


Figure 6.12: Overview of functioning of PyROOM++

The PyROOM++ VM can communicate with its environment via messages, here called *stimuli* and *response*. In addition, model execution can be monitored and controlled. For example, the user has the option to run the VM in single message modus, meaning that messages are processed one by one in sequence so that the effect of messages can be observed. The VM can be halted as well. Thanks to Python’s interpretive nature, the run-time representation can be inspected at any time. Some helper functions, for example, provide an snapshot of the current run-time model in XML (eXtended Markup Language) [BPSMM00] format.

Figure 6.13 gives an idea how a PyROOM++ session on the console looks like. The screenshot shows the Alternating Bit Protocol (ABP) [BSW69] in action. In the behavior specification, print statements help see model execution in progress.

The scheduler is invoked by the `run()` statement. We see the sender trying to transmit a “Hello” message with the alternating bit set to zero. However, the channel (realized by a typed binding) drops messages with a certain probability. The sender waits five seconds for message confirmation (the receiver sends the received message back to the sender) and retransmits the message unless the confirmation is received. In the example, the sender gets its message through on the forth attempt. The receiver’s confirmation is passed by the channel and successfully received by the sender.

A screenshot of a Windows Command Prompt window titled "Command Prompt - python -i ABPStructure.py". The window contains the following text:

```
>>> s.run()
Sender: sent message = <0, 'Hallo'>
Channel: got message = 'Hallo' at port = iCEPSender
Channel: ... and dropped it!
Scheduler: wait = 4.9943174342
Scheduler: wait = 0.00218435583292
Sender: sent message = <0, 'Hallo'>
Channel: got message = 'Hallo' at port = iCEPSender
Channel: ... and dropped it!
Scheduler: wait = 4.99079519883
Sender: sent message = <0, 'Hallo'>
Channel: got message = 'Hallo' at port = iCEPSender
Channel: ... and dropped it!
Scheduler: wait = 4.99108462109
Sender: sent message = <0, 'Hallo'>
Channel: got message = 'Hallo' at port = iCEPSender
Channel: ... and forwarded it
MessageScheduler: dataInd from ReceiveSAP<0x85d690>.ABP<0x863648>
Receiver: returned message = Hallo
Channel: got message = 'Hallo' at port = iCEPReceiver
Channel: ... and forwarded it
Sender: received message = <0, 'Hallo'>
Scheduler: wait = None
```

Figure 6.13: Screenshot of a PyROOM++ session

6.4 Improvement to High-Level Behavior Specification

Even though we excluded high-level behavior specifications in form of state machines as an explicit concern in PyROOM++, we worked on this area as well, since many protocol standards are supplemented with more or less complex state machine diagrams or state transition tables. State diagrams are a widespread technique for specifying protocols [Hol91]. For this discussion, we assume that Finite State Machines (FSM) according to the Unified Modeling Language (UML) [OMG01] are the primary means to describe behavioral aspects.

The problem we were faced with is that the behavior of several interfaces is interwoven in a single, compact state machine model; yet, it would be helpful to have a better separation of behavioral concerns: which behavior could be experienced at one interface, which one at the other interface? The segmentation of a behavioral specification according to interfaces makes (a) the behavior much more understandable and (b) allows the architect to specify behavior of one interface independently from the behavior of another interface. In return, the question is, how to properly couple and synchronize two or more state machines.

Independent of this investigation, an Ericsson internal study on the use of modeling languages for service and protocol specifications exactly points out the same problem. It shows that the coupling problem is of theoretical as well as practical relevance. It is also one of the reasons, why modeling languages like the UML (Unified Modeling Language) [OMG01] have not successfully penetrated the systems engineering domain, yet. System architects of data and telecommunication systems do not find reasonable support in today's modeling languages for their problem domain [Her99b].

6.4.1 Problem Description

For our discussion we will take the TCP protocol as an example. In chapter 3 we presented TCP on a rather detailed level. Figure 3.17 on page 98 showed the Finite State Maschine (FSM) as it is defined for the TCP service provider. For convenience purposes, the diagram is repeated here, see figure 6.14.

We already mentioned in chapter 3 that this FSM does not accurately separate the protocol part that specifies the mutual control behavior on the horizontal interface from the part that concern the vertical service interface. When we speak of horizontal and vertical interfaces, we refer to figure 3.18 on page 100: Port *t* is the horizontal interface that is associated with the Controlled Domain Model (CDM), the protocol-oriented protocol part of TCP; port *c* and *d* constitute the vertical interface. For the sake of simplicity, we treat *c* and *d* as a unity in the following.

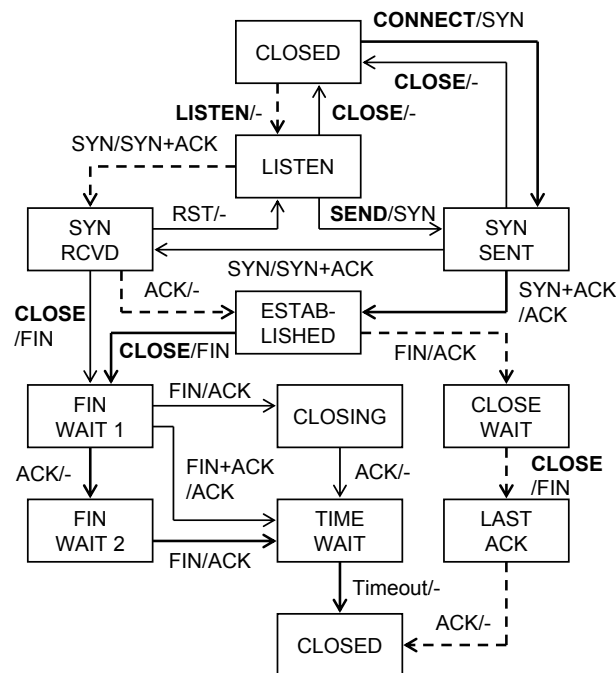


Figure 6.14: The TCP FSM figure is derived from [Tan96, p.532]. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. User commands are given in bold font.

In order to structure TCP according to its interface functions, the FSM in figure 6.14 needs to be partitioned. The result of this step is shown in figure 6.15 in UML notation.

Figure 6.15 a) displays the FSM, which corresponds in functionality to the vertical interface. Instead of using the TCP service commands LISTEN, CONNECT, SEND etc., the commands have been converted to OSI-like service primitives solely for demonstration purposes. The new set of service primitives has no equivalent for LISTEN anymore. On the server side, a connection request leads to a Conn.ind (connection indication) towards the user, who can answer with a Conn.res (connection response). Reason for this change is that LISTEN as a blocking command removes some of the problems we would like to generically discuss in the following.

Again, the client and the server side are combined in a single “vertical” FSM. From a user’s viewpoint the communication with the client/server looks like follows: When a user requests a connection (Conn.req), the client’s FSM changes to state C-PENDG. The server gets notified by the connection request via a connection indication (Conn.ind) and may respond with Conn.res, accepting the request. This is confirmed to the client via Conn.con and finally, the vertical interface be-

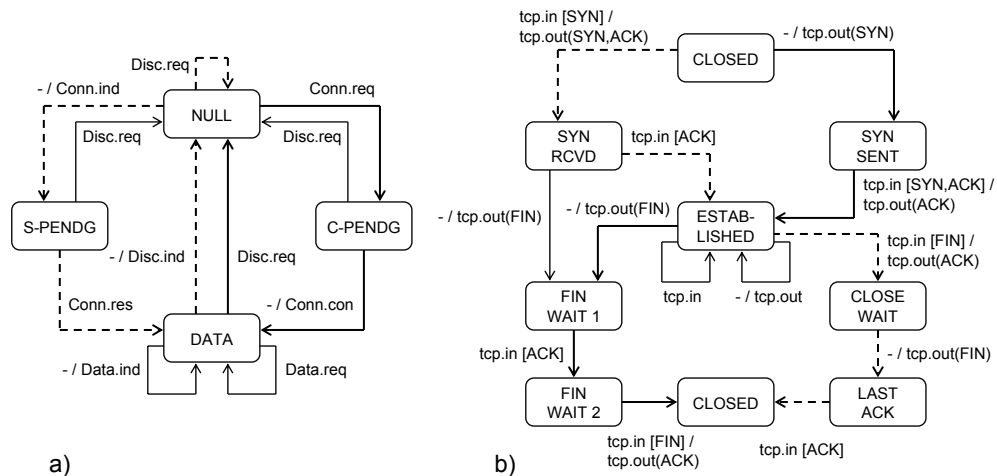


Figure 6.15: The FSMs of (a) the vertical and (b) the horizontal interface behavior of the TCP layer. The shortcuts stand for connect and disconnect; the postfixes stand for request, confirmation, indication, and response

haviors end up in state DATA. Note that neither the user of the client interface nor the user of the server interface see the underlying TCP protocol being used. They only see the horizontal interface; the layer and its use of TCP is hidden.

For the protocol protocol specification of TCP, see figure 6.15 b). Since we have not introduced any coupling yet, this FSM is strictly separated from the “vertical” FSM. That is why there is for example no indication what might have triggered the transition from CLOSED to SYN SENT at the client’s side; but when the transition is triggered, no matter how it happened, then it sends out a TCP message with the SYN bit set. Otherwise, figure 6.15 b) is similar to figure 6.14; just all the numerous details of the vertical interface have been stripped off. To reduce complexity, we slightly simplified the TCP protocol specification and added transitions to the data transfer state ESTABLISHED.

6.4.2 The Concept of Coupled State Machines

We managed to partition TCP according to its interfaces, which already is an achievement. All further details of TCP like flow control and buffering, congestion control, fragmentation, error control, window flow control etc. are hidden and subject of a refined view.

One way to couple the individual FSMs is by the usual event messaging mechanism provided by UML, that means by signals and/or call events. The drawback of this approach is that one would again tightly connect the FSMs. For example, the `Conn.req` transition of the vertical interface (see figure 6.15 a) needs to have an

activity attached that sends a signal to the horizontal interface (see figure 6.15 b). This signal would then represent the CLOSED /SYN SENT transition that triggers the tcp.out message. As a result, the “horizontal” FSM would more or less turn out to be the original TCP FSM and finally look like figure 6.14. In other words, the modeler would not be better off, and splitting of the TCP FSMs seems to be an academic exercise only.

As another possibility, the UML offers the concept of composite states, which can be decomposed into two or more concurrent substates, also called *regions*. In order to enable synchronization and coordination of regions, the UML introduced *synch states*. However, synch states do not sufficiently support enough synchronization means as the case study will show, nor do they solve the problem of synchronizing states of distinct state machines. Obviously, another technique is needed.

Our solution to this problem is the introduction of so-called Trigger Detection Points (TDPs) and Trigger Initiation Points (TIPs); both were motivated by the concept of detection points in [3GT99]. A TDP can be attached at the arrow head of an transition in a statechart diagram; it detects whenever this specific transition fires and broadcasts a notification message to all corresponding TIPs. TDPs are notated by small filled boxes, see figure 6.16. A TIP can be attached at the beginning of the transition arrow and triggers the transition to fire. An *active* TIP stimulates the transition to fire on receipt of a TDP notifier independent of the transition’s event-signature. That means, that either the event specified by the transition’s event-signature *or* the TIP can trigger the transition. Active TIPs are visualized by small filled triangles, see figure 6.16. *Passive* TIPs, on the other hand, have a locking mechanism and can be meaningfully used with “normal” transitions only, i.e. the transition explicitly requires an event-signature. The transition cannot fire unless the TIP’s corresponding TDP has been passed *and* unless the transition’s event has been received. The order of occurrence is irrelevant, it is just the combination of the TIP event *and* the transition event, which unlock the transition and let it fire. Passive TIPs behave like a logical “and” to synchronize a transition, whereas active TIPs realize a logical “or”. An example of a passive TIP can be found in figure 6.16 a); it is pictured by a small, “empty” triangle. In general, the relation of a TIP and a TDP is given by a name consisting of a single or more capital letters. Note that one or more TIPs may be related to a single TDP.

Now, the coupling of the vertical and the horizontal interface behavior can be easily described, see figure 6.16 a) and 6.16 b). For example, when a client user sends a Conn.req to the vertical interface, TDP A detects the transition NULL to C-PENDG firing and broadcasts a notifier event to all corresponding TIPs. The notifier event causes the horizontal FSM to fire the CLOSED /SYN SENT transition and results in sending out a TCP message with the SYN bit set to one; the rest of the scenario is straight forward. However, some explanations should help under-

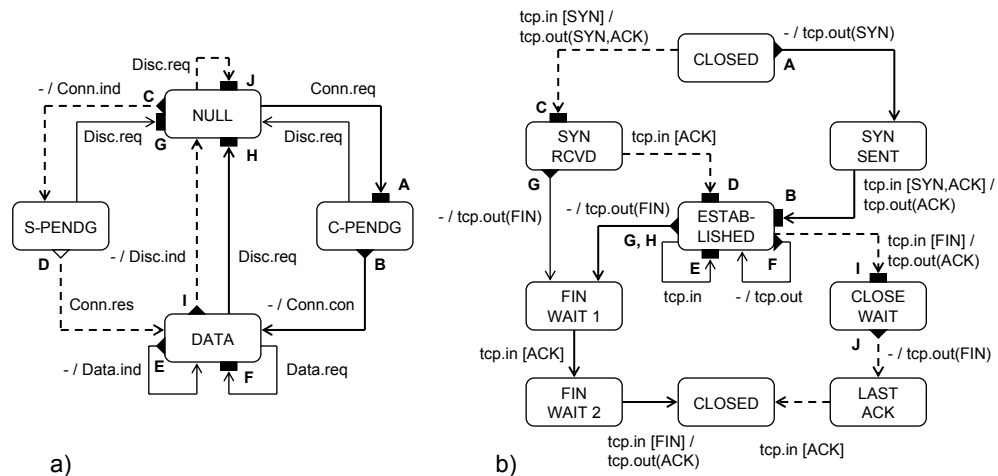


Figure 6.16: The FSMs of (a) the vertical and (b) the horizontal interface behavior of the TCP layer coupled via TDPs and TIPs

stand the purpose of a passive TIP. Let us assume, that the protocol at the server side has just entered state SYN RCVD, which triggers TIP C at the server vertical interface and results in a connection indication (Conn.ind) to the user. Now, there are two concurrent and competing threads. The user of the server vertical interface may either accept the connection indication and answer with Conn.res or, alternatively, the user may deny the request and answer with a Disc.req. Concurrently, on the protocol thread, the server’s horizontal interface enters state ESTABLISHED at some point in time. It is the passive TIP D that prevents the “vertical” FSM entering DATA on Conn.req unless the protocol has reached ESTABLISHED. On the other hand, if the user has decided to reject the connection indication (Conn.ind) via Disc.req, the horizontal interface starts the disconnect procedure based on the TDP G trigger. All this could not be done using conventional messaging without changing the FSMs.

The advantage of using TDPs and TIPs is that the FSMs remain autonomous but get coupled. They can notify each other about important state changes and use it for synchronization purposes; there is no need to introduce new event messages and modify transitions. TDPs and TIPs could be interpreted as a very special synchronization protocol, which specify FSM interaction and coordination.

6.4.3 Extending the UML

Synch states as known from the UML correspond in their behavior to what we called *passive* TIPs: A synch state is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can

enter a particular state or states [OMG01]. Clearly, synch states do not support other synchronization means between regions like TIPs do and they are not suited for inter-FSM synchronization. Good reasons to think about integrating TIPs and TDPs in the UML and to substitute synch states.

TDPs and TIPs can be smoothly integrated in an event driven execution model for FSMs. The prototype we developed at Ericsson (programmed in Python [Lut96]) treats TDPs as a specialization of messages, see figure 6.17, and dispatches notifier events to the event queue. The implementation of TIPs required only a few modifications to the event processor.

If one compares the prototype design and the metamodel for state machines (see section 2.12 of the UML [OMG01] semantics), the required extensions to the UML can be easily identified: First, the notifier event needs to be subclassed to the event metaclass;³ this can be achieved by using stereotypes. Then, it is to be decided how TDPs can be attached to the transition metaclass. Since transitions are restricted to have not more than one event trigger, it is not possible to add TDPs as a second trigger. Rather, the transition metaclass can be extended by some few properties. A TDP property is needed referring to the notifier event, optionally added by a property holding a list of state machines the notifier event is selectively broadcasted to. Another property are the TIP and the TIP type, which hold the notifier reference and the value *active* or *passive*, respectively. The required changes to the execution semantics of state machines are uncritical, since the UML is relatively open to adaptations. To conclude, the extensions described are the simplest form to introduce TDPs and TIPs to the UML using its extension mechanisms [HvW99].

Note that TDPs and TIPs make synch states superfluous. TDPs/TIPs contain the concept of synch states but allow much more semantic variations and extensions. Synch states are an oddity in the UML with no clear conceptual roots; TDPs and TIPs are their generalization but they are put in a meaningful semantical context of transitions and events. In fact, TIPs and TDPs specify a synchronization protocol between states machines or regions. Such a protocol does not only seem more appropriate to capture complex interactions of synchronization but also semantically cleaner. That is why we propose to remove synch states from the UML metamodel and instead introduce the notifier event subclass, insert metaclasses for TDPs and TIPs and associate them to the transition metaclass. This would enable flexible semantic extensions via stereotypes to the UML user.

The prototype we developed to show the functioning of TIPs and TDPs used the TCP example. The screenshot, see figure 6.18, shows a simple GUI that helped monitor FSM execution in single-step modus, modify message parameters and observe the state changes.

³Regarding events, the UML is a bit different designed than our message based prototype.

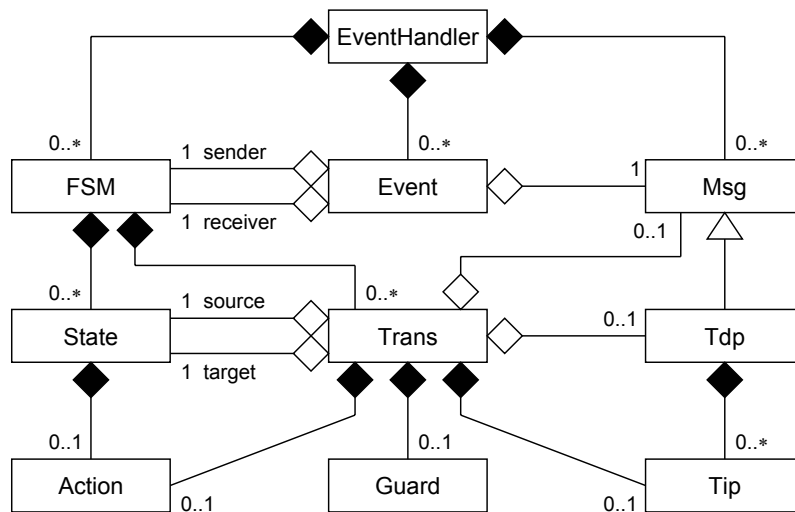


Figure 6.17: The design of the coupled FSM prototype

6.4.4 Conclusion

Actually, the TDP/TIP concept relates very much to the observer pattern [GHJV95]; it allows the modeler to notify other FSMs about state changes. Because of the distinction in active and passive TIPs, the concept of coupled state machines implements an extended observer pattern. This lifts the observer pattern from its use in the design domain in form of class diagrams to the modeling domain with an explicit notation for coupling, which is a quite interesting aspect. Furthermore, it is an interesting question, if TIPs and TDPs could be of use in sequence diagrams or Message Sequence Charts (MSC) [ITU99c].

Since the approach presented gives means to specify and separate aspects of a modeling entity, one could also investigate to which extend TDPs and TIPs enable aspect-oriented modeling in extension to aspect-oriented programming [KLM⁺97]. It also allows the modeler to specify APIs (Application Programming Interface) much more elegant; for instance, the TCP vertical interface could be seen as an API to TCP. As was shown in the case study, the design of communication protocols gains a lot of clarity from the separation of logical concerns. In short, it looks like that many application areas could benefit from using coupled state machines.

Due to the specific nature of the application domain (data and telecommunications) we study, we cannot claim that we have identified all types of TDPs and TIPs required for coupling FSMs in an efficient manner. Extensions or specializations are conceivable. However, TDPs and TIPs appear to be a powerful modeling concept, they substitute synch states, and put a modeler in a better position especially for modeling the coordination and synchronization of concurrent systems.

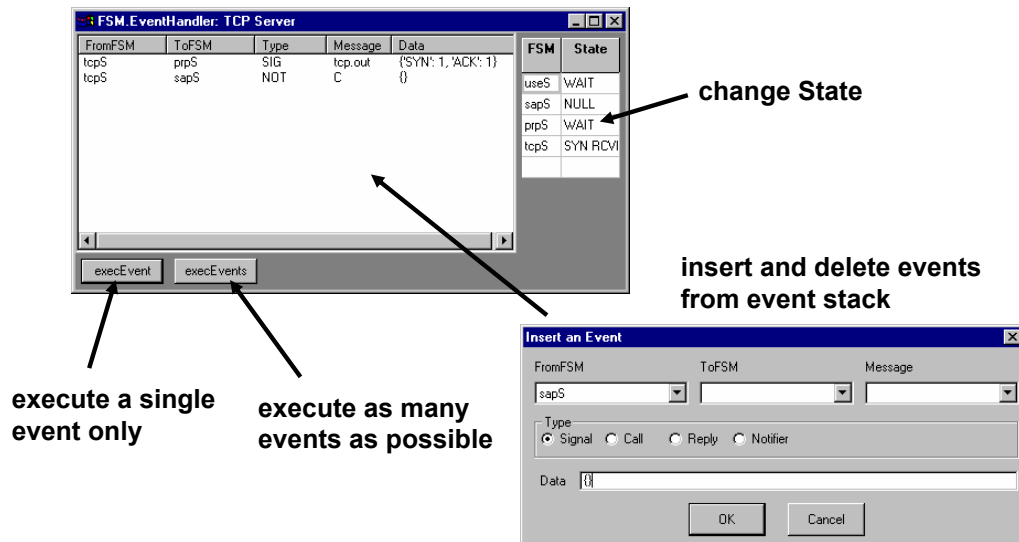


Figure 6.18: Screenshot of the coupled FSM prototype

6.5 Summary

The formal fixation of all the improvements and enhancements to ROOM have been subject of this chapter and resulted in the specification of the ROOM++ language in form of a meta-model including well formed rules. ROOM++ is a superset of ROOM with the additional features we identified as necessary but missing in ROOM for system architecture modeling. In specific, the design of the ROOM++ language contrasts to the design of ROOM by

- the unification of the port/SUP/SPP concept;
- the option to declare a port as a control or data port;
- the possibility to explicate the Controlled Domain Model (CDM) of a control port
- the option to specify an address type, an address list and an address topology for a port; thereby the modeling of address spaces is supported
- the option to relate ports to functional spheres of purpose, called planes
- the relaxation of port semantics, so that internal end ports are not required anymore and conjugation symmetry is permitted for the endpoints of a (typed) binding
- the introduction of the concept of a typed binding, which enables the modeler not only to use n-ary connectors but also to assign behavior to a binding
- the removal of the equivalence concept
- the option to use a classification system for actor classes

Together, this set of language features gives means for very expressive architecture models. Due to a very careful language design, these features do not make the design of ROOM++ more complicated than that of ROOM. Rather the opposite is the case. Because of a careful analysis of ROOM's notion of layering, the SPP and SUP concepts, which make up a considerable part of ROOM's meta-model, could be integrated in the port concept. The design of ROOM++ benefited from these insights.

ROOM++ has been implemented in the programming language Python, that is why the implementation is called PyROOM++. Python has been also used as the action language to "specify" an actor's behavior component. Thereby, we could drop the need for a high-level behavior specification formalism, such as finite state machines. Behavior is fully defined via Python programs.

Nonetheless, we studied the use of finite state machines for high-level behavior specifications. The language of reference for state machines was the UML. The problem we were faced with concerned the wish to independently specify the functional behavior of interfaces, which requires to retroactively add the coordination and synchronization of the parts to make up a whole. For that purpose we invented the notion of Trigger Initiation Points (TIP) and Trigger Detection Points (TDP). TIPs and TDPs are special concepts to specify a synchronization protocol for the coordination of related state machines. A proof of concept was realized by an implementation of a prototype.

Chapter 7

Methodology

In this chapter we wrap up the systematic approach we aimed for in this work and which has been already manifested in the previous chapters. And last but not least, we come back to the case study we introduced at the very beginning of this work in chapter 1. We apply our methodology and ROOM++ on the case study and hope to persuade the reader that the result is much better than the sort of architecture models we find in the standards.

In section 7.1 we unroll our methodology in several *method blocks*, each method block being a self-contained methodological unit. All units together make up the complete methodology. Method block *system network architecture* is the core unit the other blocks can be plugged-in. In section 7.2, we revisit the SIGTRAN case study and show, how the SIGTRAN architecture looks like when we approach the modeling process systematically and use ROOM++ as a modeling language. Our experiences with using the methodology and ROOM++ at Ericsson are reported in section 7.3. Section 7.4 summarizes this chapter.

7.1 Systematic Approach: From Standards to System Architectures

When we say that we present “a systematic approach to create logical models of network architectures” (page 5) we mean that we take a standard or an architectural problem and (a) transform this problem into a specific kind of representation and (b) do that transformation process in a certain way. The transformation process is hardly to automate and requires a thoughtful strategy. A methodology supports the transformation process by a set of guiding questions, recommendations, heuristics of action and solution patterns.

In our case, the new form of representation uses ROOM++ as a formal and visual language. The borderline between ROOM++ as a language and as a supporting part of the methodology is not precisely to draw: ROOM++ has been adapted to the needs of our systematics. Issues that would be a methodology matter in ROOM, are now an integral part of ROOM++. Take for example addressing. In ROOM, modeling address spaces would require a very strict and detailed method description of how to realize addresses inside the specification of a behavioral component. In ROOM++, one barely needs methodic advice on how to model address spaces, since the language supports address types, address lists and address topologies.

A method is to some extent always a compensation of the deficiencies of a bad or improper language. For example, the UML without a lengthy description of *how* to use it for modeling a telecom system architecture is almost useless. However, ROOM++ is well adapted to that domain, the architect will tend to use it purposefully almost naturally and needs a much shorter *how-to* manual.

As was mentioned in chapter 2, we presume a certain view on the subject matter of communication systems; namely, we presume a reference framework/model that pre-structures our problem domain of standards and system architectures. Otherwise we are without any orientation and without any terminology to categorize, structure and navigate through the problem at hand. System engineering and engineering in general starts first with identifying and classifying system elements and their structural relationships, be they static or dynamic, and then goes on to “juggle” with these elements and relationships. The framework we have chosen as an engineering basis is a generalization of the OSI and TCP/IP reference model, because most standards adhere to either OSI RM or TCP/IP RM. Often, standards explicitly mention their compliance to one of these RMs.

The new, generalized reference model for system network architectures is described in section 7.1.1 under “Setting the Scene”. This section is at the heart of our methodology. Refinements and details to the method core are added by section 7.1.2, section 7.1.3 and section 7.1.4.

Our systematics is broken up in *method blocks*. One block concerns the overall analysis and construction of a *system network architecture*, others concern *protocol entities*, *resource entities*, and *aspect entities*. A user may start with any appropriate method block. In each method block, we first start “Setting the Scene” with a rough conceptual schema of the conceptions we are dealing with. Then, we go on and provide a list of questions and actions in the “Method Part”. The instructions provided there should help mainly in analyzing standards. Then, in “Using ROOM++” we show patterns of use we elaborated throughout this work for this specific method block. Finally, in “References” we provide some hints, where in this work we laid the foundations for this part of our methodology.

7.1.1 Method Block: System Network Architecture

Setting the Scene

The conceptual schema for the analysis of communication networks is shown in figure 7.1. This schema applies for analyzing system and network architectures, architecture frameworks and standards. Communication Service and Communicating Entity are highlighted because of their importance in our methodology. For the theoretical background of this approach, see chapter 4 and chapter 5.

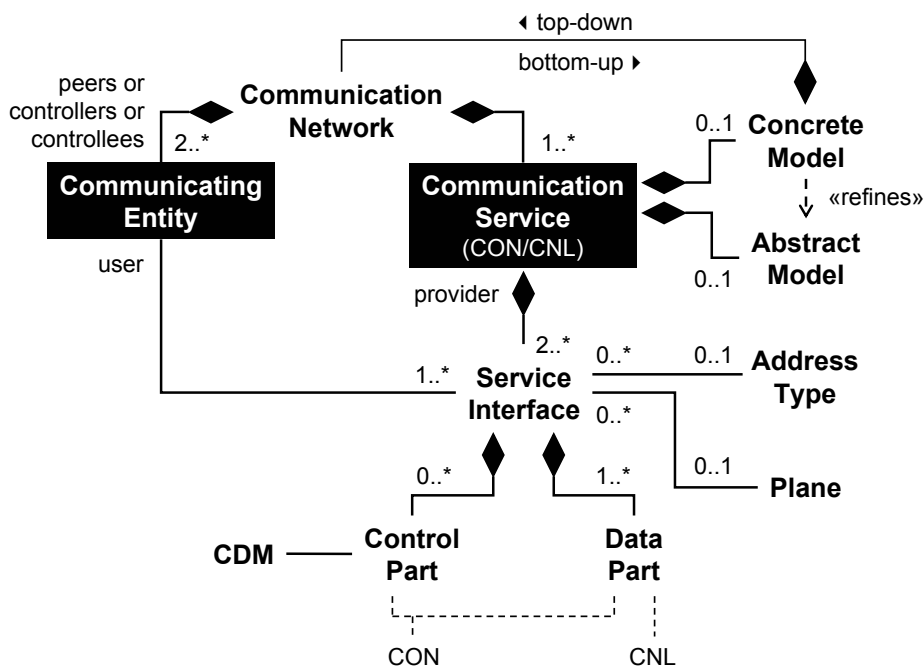


Figure 7.1: Conceptual schema for system architecture

This conceptual schema concerns the overall organization of a system architecture as a system network architecture. The key elements of a network architecture are two or more *communicating entities* and one or more *communication services*. These two elements make up a *communication network*. The communicating entities may act in different sorts of roles: they may be *peers*, *controllers* or *controllees*. As a rule of thumb, controllers and controllees can be distinguished by a clear control-oriented communication relationship, whereas peers cannot. Among peers, a control-oriented communication style may be temporal but the roles of who exerts control may change dependent of the communication context.

A communication service is the provider of a *service interface*; a communicating entity is the user of a service interface. The service interface can be usually decomposed in *control parts* and *data parts*. In case of a *connectionless* (CNL) communication service, the service interface consists only of a data part; for a *connection-oriented* communication service, the service interface has in addition a control part. For the control part, a *Controlled Domain Model* (CDM) is associated to it. The service interface may have an *address type* associated to it and may have a *plane* attribute.

Key to our method is that the communication service can be modeled as a very *abstract model* and/or as a more *concrete model*. One model may be substituted by the other model. The concrete model itself represents a communication network; herewith, our conceptual schema becomes circular. Since the circularity can be used either for *top-down* or *bottom-up* design of the architecture or for a combination thereof. In any case, the concrete model has to fulfill the conditions of *communication refinement* with regards to its “upper” communication service as discussed in chapter 5. In practice, the circularity is not infinite; it ends at communication services, which only have an abstract model and no concrete substitution. It is important to note that the condition of communication refinement imposes constraints on the design of the communicating entities, which is not visible in the conceptual schema! For more details, please refer to chapter 5.

The whole set of elements of a communication network, starting with the “top-most” communication network, and the way of grouping these elements are regarded as the *system (network) architecture*.

Method Part

- First, identify the constituting elements of a communication network: Which are the communicating entities? What kind of communication service do they use?
- Which roles do the communicating entities have? Heuristic: If they belong to the same functional sphere (plane) in the network architecture, they are

most likely peers; if they belong to different planes, a control-oriented relationship could determine the roles.

- Get a clear picture on the service interface and define the message schema. What kind of message schema does the user need, what kind of message schema does the provider offer? Hint: For a certain architectural purpose it might be sufficient to simplify the parameters for messages, work with symbolic values, assume some parameters as fixed etc. Architecture modeling is about simplification to the degree that the main behavioral features of the architecture remain demonstrable by model execution. Distracting details should be avoided if feasible. Note: Because of the recursive nature of the conceptual schema, the message schema of an “upper” communicating entity can be possibly derived from the schema provided by a “lower” communicating entity and vice versa.
- Distinguish between service messages for control and service messages for data transfer. Split the service interface accordingly. Hint: If the service interface offers a connectionless service, there are only data messages; if the interface offers a connection-oriented service, there must be control messages.
- If there are control messages: Define the underlying CDM.
- If the communicating entity is a protocol entity, a resource or an aspect entity, please refer to the corresponding method blocks. If not, the communicating entity is supposed to be an *application*; an application is not subject to further design recommendations.
- If the communication service represents one or more address spaces, define suitable data classes as address types and relate these types to the corresponding service interface. Make up your mind about the address topology and design a model for it if meaningful. Hint: Sometimes there are options to model an address type as a complex data class or as a cascade of address spaces. Cascades of address spaces can be modeled as aspect entities, see method block *aspect entities*.
- Addressing might require the specification of registration messages. Check, if there is a need for registration messages and add them to the message schema.
- Determine the plane attribute of the service interface if of interest. Recommendation: For complex architectures it might be a good idea to defer the

decision to assign planes to interfaces until the architecture is reasonable complete.

- Define an abstract and/or a concrete (ideally executable) model for the communication service. If there is no concrete model, the abstract model is a must. There are no constraints put on the design of the abstract model, we just recommend patterns, see below.
- The concrete model breaks down to another communication network. Repeat the steps of this method part. Regarding the consequences of this break down (called layering in telecommunications), see below.
- Ensure that the address types modeled for the abstract model and the cascade of address types of the refined model remain consistent.
- If wanted, define also an executable model for the communicating entities.

The circularity in the concept schema puts an implicit requirement on communicating entities, which are part of a concretion of an “upper” communication service. These entities have to provide the service interface in place of the resolved, “upper” communication service but are still users of a service interface provided by the communication service on their communication network level. If this may sound cryptic, remember the abstraction hierarchy in chapter 5: the communicating entities in the abstraction hierarchy have an “upper” and an “lower” interface – it is exactly that, what we are after. If you look at method block *protocol entity* and method block *aspect entity* you will see that both are suitable variants of a communicating entity, providing a service interface to the “higher” levels and a protocol and a communication interface, respectively, towards their communication service. Resource users and resource providers are no exemption in this respect, they are just methodically analyzed differently, see method block *resource entity*.

The use and the understanding of the abstraction hierarchy is key to the way we relate communication networks to each other. In addition, the design approach, be it node-centric or network-centric, enforces a certain way of grouping elements and shapes the outlook of the final architecture. The basic grouping scheme is outlined in figure 7.2. For more details, please refer to chapter 5.

So, a final item in this method block is:

- Decide on how to group the communicating entities and the communication service of a system architecture. Heuristic: For an implementation viewpoint, node-centric grouping is likely to be of use; otherwise, network-centric grouping is recommended.

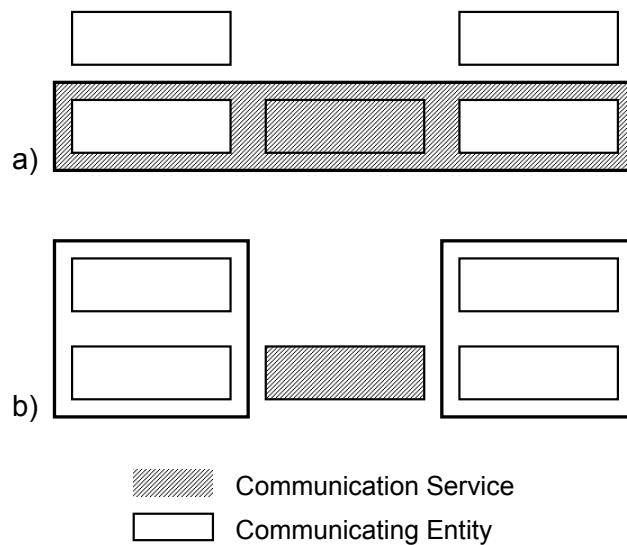


Figure 7.2: Grouping of communication entities and the communication service: a) network-centric, b) node-centric

Using ROOM++

In ROOM++, communicating entities are modeled as casual actor classes. Communication services are also modeled as actor classes but we tag the actor class with a category value. Reason is that we allow communication services to be represented by either the actor class or a typed binding having the actor class as the binding's type. The latter variant is strictly in line with our algebraic reasoning about distribution and with the introduction of a complex connector. The former variant is a helping alternative, especially for modeling languages, which do not support the notion of a complex connector like e.g. ROOM.

For the specification of the “ingredients” of communicating entities we do not provide any methodological advise. It is up to the modeler to make intelligent use of all ROOM++ language features; this requires some experience and practice with ROOM(++).

All other issue like address types, address topology, control ports, data ports, and the CDM are supported by ROOM++ as language features. Their use should leave no questions. Remember that the control-port is assigned to the user of the service interface not to the provider.

For connection-oriented and for connectionless communication services we can offer two patterns for the abstract model, shown in figure 7.3. Slightly simplified versions of these diagrams were discussed in chapter 4, see figure 4.24 on page 156.

Note that in figure 7.3 a) the data port has been moved inside the communi-

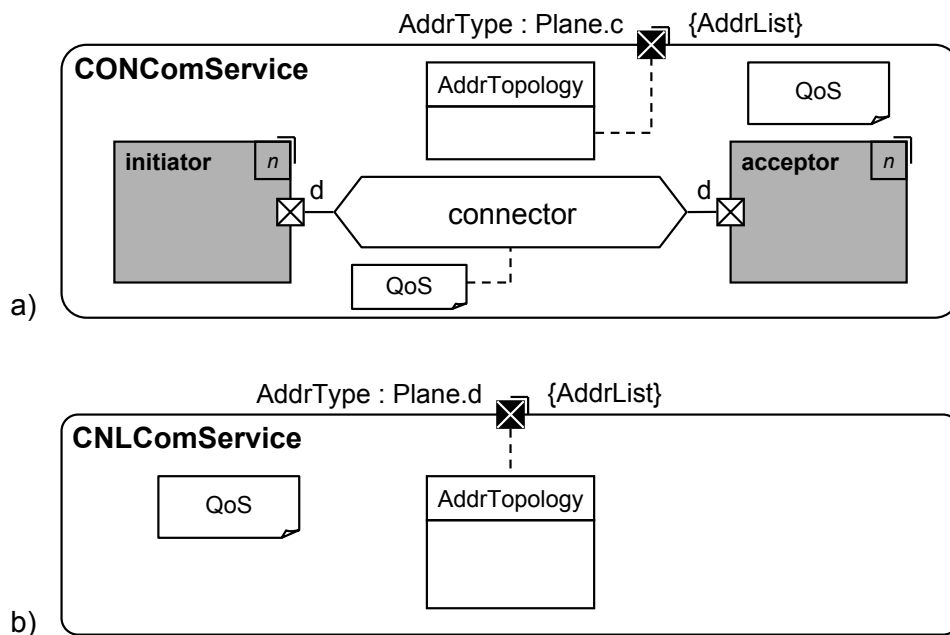


Figure 7.3: Abstract model patterns for a (a) connection-oriented (CON), (b) connectionless (CNL) communication service

communication service via the import actor reference. Communication service interfaces must not necessarily appear on the border of a communication service.

Node- and network-centric design approaches differ in the way a container actor class is put “around” the communicating entities and the communication service. The patterns that apply are shown in figure 7.4. Here, the communication service is realized by a typed binding. Note that in case of a node-centric communication refinement, the vertical interfaces usually change their kind to SIP, resulting in a SPP and a SUP instead of “casual” ports.

References

Chapter 4 is completely devoted to modeling networked communication. There, the reader will find a very detailed introduction into this part of the methodology including the above mentioned patterns. Also addressing is explained. Examples refer to the design of UDP and TCP on a user and on a provider level. Alternatives on how to model the CDM are described in chapter 3.

The abstraction hierarchy and nesting of communication network by means of communication refinement is subject of chapter 5.

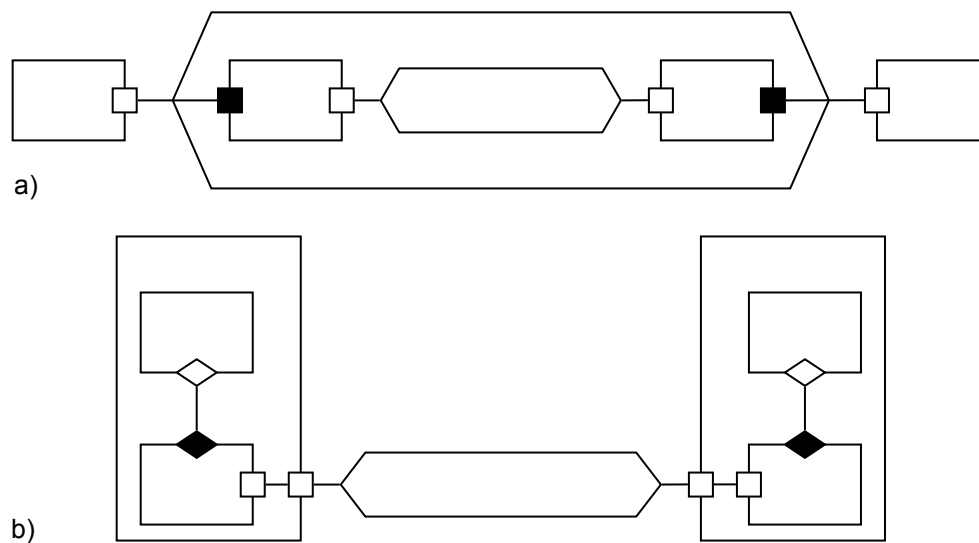


Figure 7.4: Pattern for communication refinement: a) network-centric, b) node-centric

7.1.2 Method Block: Protocol Entity

Setting the Scene

The conceptual schema for the analysis of protocol entities is shown in figure 7.5. This schema applies for analyzing a protocol specification. Most standards in (tele)communications concern a specific protocol and usually present the protocol from the viewpoint of an entity that realizes the protocol. This viewpoint is taken here. We exclude protocols that define remote resource control, like MGCP [CGH⁺00], and refer to method block *resource entity*. For background and reasoning of this approach, please refer mainly to chapter 3.

A *protocol entity* consists of a *service interface* and a *protocol interface*. The service interface has a *data part* and – if offering a connection-oriented service – also a *control part*. For the protocol interface it is characteristic that the interface has either a *control part* or a *data part*. Control parts are associated with a CDM. The service interface may have an *address type* and a *plane* attribute.

Method Part

- Specify the message schemata for the service interface and the protocol interface. Hints: The message schema for the protocol interface is usually very simple; here, the aim is to make simplifying assumptions about the message parameters, if possible. For the service interface, a specification is usually given in the standard; the task often is to just convert the standard's

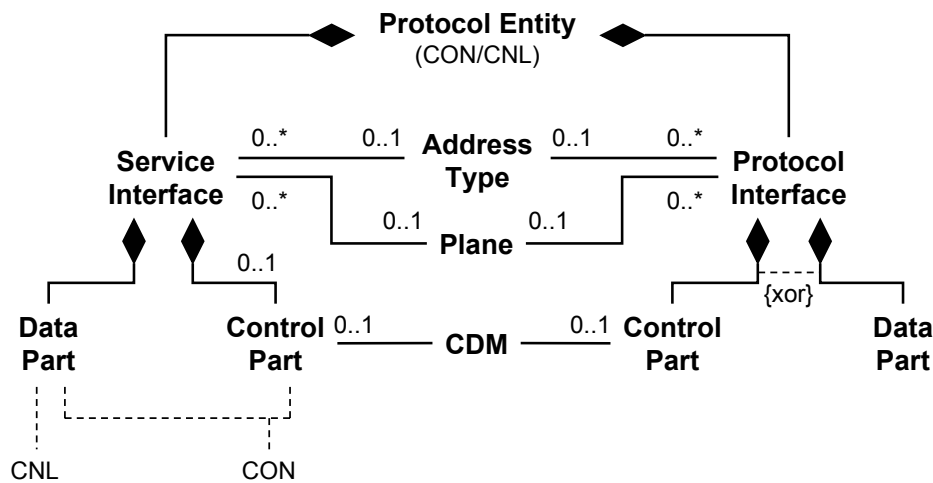


Figure 7.5: Conceptual schema for protocol entity

specification to a message-based communication paradigm.

- For the control ports, define the underlying CDM. Hint: For the protocol interface the CDM basically is the state machine as defined e.g. in a standard. However, most standards do not separate the state machine for the service interface and the state machine for the CDM. Here, we would like to point to the section about coupled FSMs, see section 6.4 on page 226 ff.
- Specify an address type for the interfaces. Heuristic: Usually, in the protocol specification, the addressing schema is explicitly given or assumed so that an address type is easy to define for the service interface. For the protocol interface, an address type may be not meaningful to define.
- Determine the plane attribute of the interfaces if of interest.
- Specify a working, executable model of the protocol entity and plug them in the context of a communication network, see method block *system network architecture*. Recommendation: The level of detailed should be determined by the demonstration purpose of the architecture model.

Using ROOM++

Protocol entities are modeled as actor classes. Again, the full set of techniques offered by ROOM++ is available to the modeler for the internals of the entity; we do not provide any methodological advise here. According to the conceptual schema, a model of the protocol entity adheres to one of the four following patterns on a coarse grain level, see 7.6.

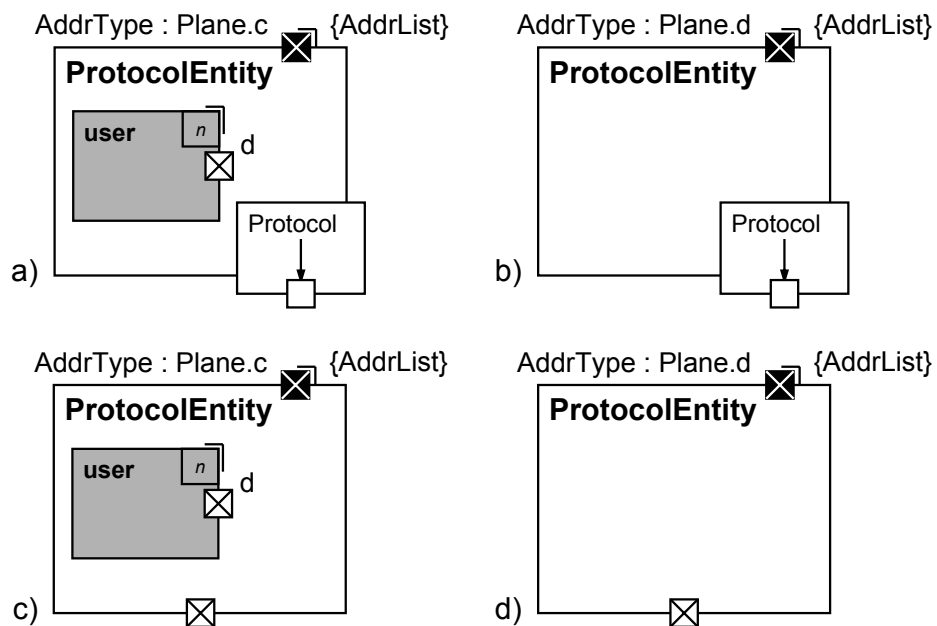


Figure 7.6: Patterns for a model of the protocol entity; a)-d) represent variations of service and communication interface

Figure 7.6 a) is a typical pattern of a protocol entity offering a connection-oriented communication service (the control port is external, the data port so to speak “internal”) and specifying a protocol-oriented protocol on the communication interface. Many protocol specifications that provide a connection-oriented service interface can be subsumed under this pattern. Figure 7.6 b) shows a less common pattern: a connectionless service interface and a protocol-oriented protocol. Figure 7.6 c) is also quite unusual: a connection-oriented service is realized by a data-oriented protocol. Another wide-spread pattern shows figure 7.6 d): a connectionless service is realized by a data-oriented protocol. Many protocol specifications that provide a connectionless service interface can be subsumed under this pattern.

References

The foundations for protocol-oriented protocols and data-oriented protocols were laid in chapter 3. Examples of pattern a) and d) are the TCP and UDP provider level as presented in chapters 4 and 5.

7.1.3 Method Block: Resource Entity

Setting the Scene

The conceptual schema for the analysis of resources is shown in figure 7.7. This schema applies e.g. for analyzing distributed resource access and resource protocol specifications. The schema is also valuable for non-distributed resource providers and users, but that goes beyond the scope of our methodology. For background and reasoning of this approach, please refer to chapter 3 and chapter 5.

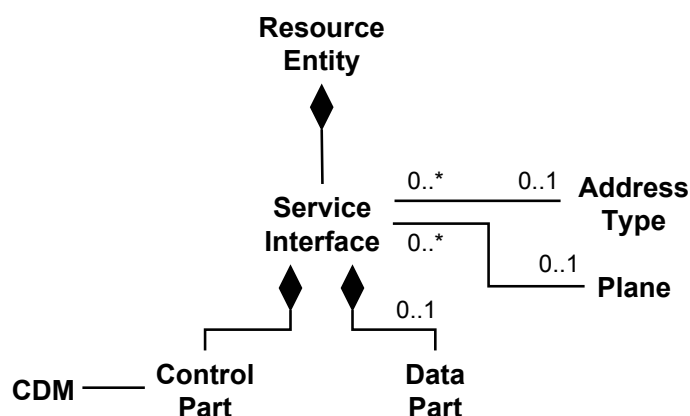


Figure 7.7: Conceptual schema for resource entity

A *resource entity* can be accessed via a *service interface* that must have a *control part* and may have a *data part*. If a resource protocol is given, the protocol is a specification of the resource's service interface. The control part is associated to a CDM. The service interface may be associated to an *address type* and a *plane* attribute. The party that accesses and uses the resource entity is called resource user, the party providing the resource is called resource provider.

Method Part

- Specify the message schema for the resource's service interface. If a resource protocol specification is given, the aim is to (a) adapt the protocol specification to a message-based communication paradigm and (b) simplify the message parameters to the demonstration purpose of the architecture model.
- Specify the CDM for the control part of the message schema.
- Make yourself clear about, who is the resource user (the controller) and who is the resource provider (the controllee). This is important for method

block *system network architecture*. Heuristic: Very often, resource access is regarded as vertical communication.

- If meaningful, define an address type and a plane attribute for the service interface.
- Specify a working, executable model of the resource. Recommendation: The level of detailed should be determined by the demonstration purpose of the architecture model.
- Embed the resource in the context of a resource provider. In addition, design an executable resource user. Plug-in the resource provider and the resource user as communicating entities in a communication network context, see method block *system network architecture*

Using ROOM++

A resource user has at least a control port and a CDM, whereas the resource provider only has one or more data ports. A simple pattern is shown in figure 7.8.

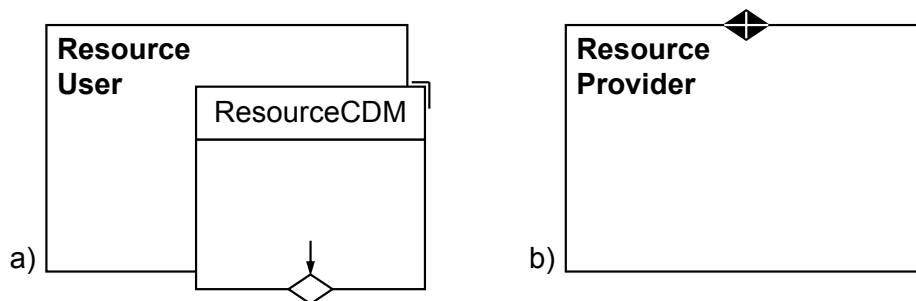


Figure 7.8: Simple patterns for a) resource user and b) resource provider

Note that in the context of a communication network it is assumed that resource user and provider are remote to each other or planned for distant communication and are therefore attached to a communication service.

References

We laid the basis for resources and resource control in chapter 3 and exemplified it on MGCP. See also chapter 5.

7.1.4 Method Block: Aspect Entity

Setting the Scene

The conceptual schema for the analysis of an aspect entity is shown in figure 7.9. This schema is a generalization of a resource and protocol entity and applies to all sorts of communicating entities including the application level. Aspects were a late topic in chapter 5, see section 5.3.3 on page 186 ff. Nonetheless, they are a very interesting addition to our methodology and broaden the applicability of our approach considerably. We explicitly would like to stress that aspect entities are also subject to the constraints of communication refinement.

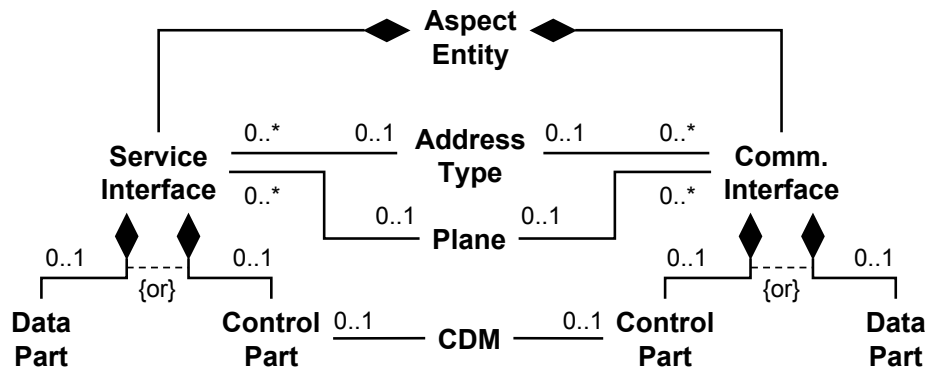


Figure 7.9: Conceptual schema for an aspect

An aspect entity is a communicating entity (see method block *system network architecture*) that is plugged-in in a communication network context, where the communicating entity has to fulfill an “upper” *service interface* and demands a *communication interface* towards a communication service. An aspect entity may have any arrangements of *control parts* and *data parts* of its interfaces. As known from the other method blocks, a CDM is associated to the control part, and an *address type* and a *plane* can be associated to any of the interfaces.

Method Part

Since aspects are usually not standardized, nothing can be used as a given description of the aspect. The definition of an aspect entity is very much a matter of the creativity and experience of a modeler.

- Specify the message schemata for the service interface and the communication interface.
- Define the CDMs for control ports.

- If meaningful, specify the address type for the interface.
- Determine the plane attribute, if of interest.
- Specify an executable model of the aspect entity.
- Plug-in the aspect entity as a communicating entities in a communication network context, see method block *system network architecture*

Using ROOM++

An aspect entity is modeled as an actor class. In the course of this work, one aspect we defined was the *demultiplexer* – even though we did not call it an aspect when we introduced it. When we looked closer on MGCP, we found *connection management* to be an aspect entity as well. Both aspect entities are repeated in figure 7.10. We may expect to find more aspects for certain kinds of networks and ways of stratifying a system architecture.

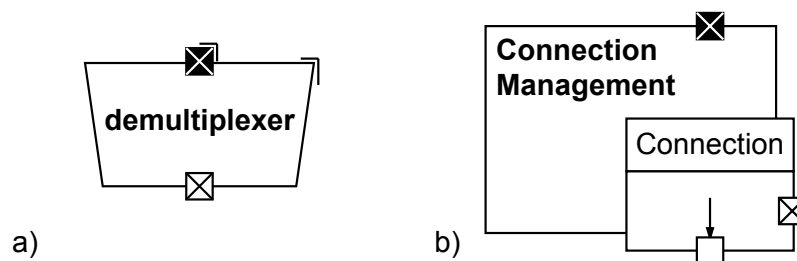


Figure 7.10: Two aspect entity patterns: a) demultiplexer b) connection management

Especially for aspect entities, the category attribute for actor classes may be meaningful to classify aspects and structure them.

References

Aspects were a byproduct of our investigation on remote resource access exemplified on MGCP, see chapter 5. Aspects are a welcome generalization of the notion of a protocol entity and of a resource.

7.2 The Case Study Revisited

The case study on SIGTRAN (SIGnaling TRANsport) presented in chapter 1 is of some complexity. One has to read and understand numerous standards that concern IP technology on one hand, and Signaling System No. 7 (SS7) technology on the other hand. Table 7.1 gives an overview of the minimal set of standard documents required to understand SIGTRAN and the model we present below. We guess that the number of pages to be read in this case is also a good guess for the amount of information a system architect typically has to explore and understand for a certain assignment. Of course, the level of experience, background and knowledge make a substantial difference, whether about 1000 pages are regarded as much or little.

Table 7.1: Standards covering case study

Standard	Title	Pages
H.248 [ITU00]	Gateway Control Protocol (GCP)	115
Q.700 [ITU93b]	Introduction to Signaling System No. 7 (SS7)	24
Q.701 [ITU93c]	Functional description of MTP	26
Q.703 [ITU96a]	Signaling link	95
Q.704 [ITU96b]	Signaling network functions and messages	217
Q.705 [ITU93d]	Signaling network structure	28
Q.706 [ITU93e]	Signaling performance	41
RFC2719 [ORG ⁺ 99]	Framework Architecture: Signaling Transport	24
RFC2960 [SXM ⁺ 00]	Stream Control Transmission Protocol	134
RFC3332 [SMPB02]	MTP3 User Adaptation Layer (M3UA)	120
		824

No methodology can bear the system architect's burden to digest all this technical information. Yet, a methodology and a supporting modeling language can very much help dealing with this large amount of information, to structure, to condense and abstract it. A methodology helps in processing information by triggering a reflection process on what has been read and how it could be structured.

In the following, we have to omit a lot of details and make some simplifications; there is much more information in the standards than can be condensed in half a dozen of diagrams. But that is what architecture modeling is about: we simplify and neglect whatever can be dropped for the *purpose* of the architecture model. Here, the purpose is to give the reader a fairly well understanding of how SIGTRAN solves the problem of conveying signalling messages over different

means of message transport. Our SIGTRAN architecture model is correct in the sense that any implementation of SIGTRAN should fulfill the model.

Note that the case study is a real-life example. At Ericsson, we used the method and a simplified ROOM++ notation in order to understand SIGTRAN and to propose architectural alternatives.

7.2.1 Overview

To support the reader with a navigation help through our process of step-wise applying the methodology and breaking down the model into more and more model artifacts, we provide an overview map in figure 7.11. Since we chose to go top-down, the result of the first step is a model of the top level communication network. The communication service is then made concrete by another communication network in step 2. Next, the interna of the communication service of step 2 are decomposed into three model artifacts. Step 3 and step 4 detail the respective communication services. In step 5, the remaining artifact is refined into a communication network of its own right. Step 6 and step 7 are not visible in the figure. In step 6, we show how a node-centric view on SIGTRAN looks like. In step 7, the model is evolved further.

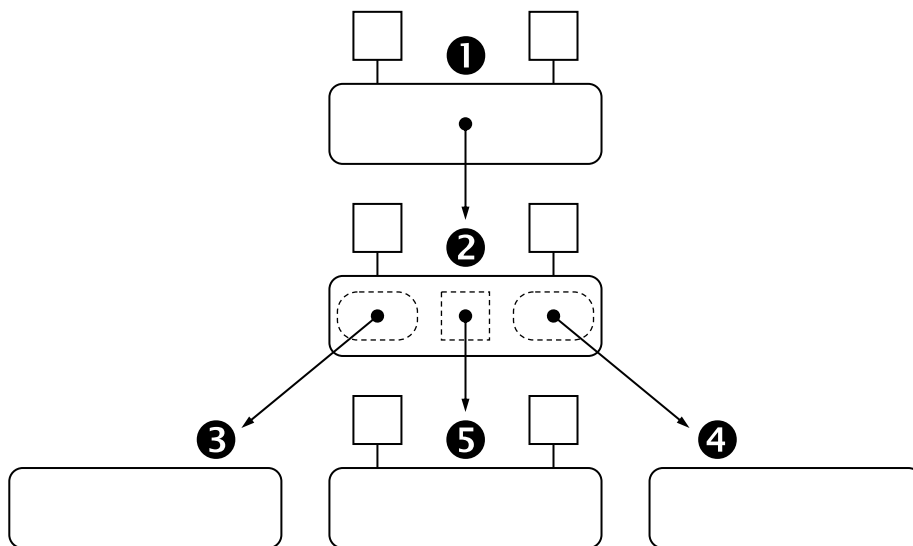


Figure 7.11: Overview of SIGTRAN modeling process

Of course, this overview is an *a posteriori* reflection on the steps taken and is offered as a service to the reader. Naturally, there are alternative choices in the process of applying our method. However, the result of the following breakdown

is guided by our interpretation of the SIGTRAN standard. Other choices may lead to a system architecture model that is not compliant to SIGTRAN.

7.2.2 Step 1: MTP3-User Communication Network

The primary goal of SIGTRAN is to make the transfer of signaling messages independent from its means of transport. Traditionally, signaling relies on an infrastructure called Signaling System No. 7 (SS7). So called User Parts (UP) make use of a Message Transfer Part (MTP) in order to convey messages. MTP is decomposed into three layers, namely MTP1, MTP2 and MTP3, with MTP3 being the “topmost” layer offering a connectionless communication service to the user parts. Each MTP layer implements a protocol for communication to another, distant peer.

The aim of SIGTRAN is not to change the user parts and the interface towards MTP3, that is too much of a valuable legacy, but to add flexibility to the transport part. Different means of transport should be possible and peacefully coexist. Besides the traditional means of transport via MTP layers, the SIGTRAN standard proposes means to transport MTP3 protocol messages on the basis of IP technology. Furthermore, SIGTRAN is very much concerned about smooth interaction between different transport technology domains. Messages should easily pass the border from one transport domain to another domain and vice versa. The user parts should notice no difference whether they communicate to another user part based on the same or another transport technology. In fact, the message transfer part should be completely transparent to the users.

This architectural vision of SIGTRAN can be easily modeled top-down if we start on the level of user parts and ask – according to our methodology – the questions: “Which are the communicating entities?” and “How does the communication service look like?” The resulting communication network is shown in figure 7.12. The user parts are called MTP3Users here.

To make the architectural vision of SIGTRAN clear, we distinguish two groups of users. On the left hand side we group the user parts, which are based on SS7 technology; on the right hand side we group the user parts, which are supposed to be based on IP technology. That little difference is indicated in the bottom by the remarks “SS7 domain” and “IP domain” in italics. Besides, there is no qualifying difference between the user parts, meaning that all user parts make use of the MTP3 service protocol and adhere to the same address type.

Despite all the complexities of SS7, the interface provided to its users is quite primitive. In our model it is the MTP3UComService that provides the communication service to its users. The service provides connectionless communication: messages can be sent via MTP-Transfer.Req and received by MTP-Transfer.Ind. “Req” stands for “request” and “Ind” for “indication”. For details on the message

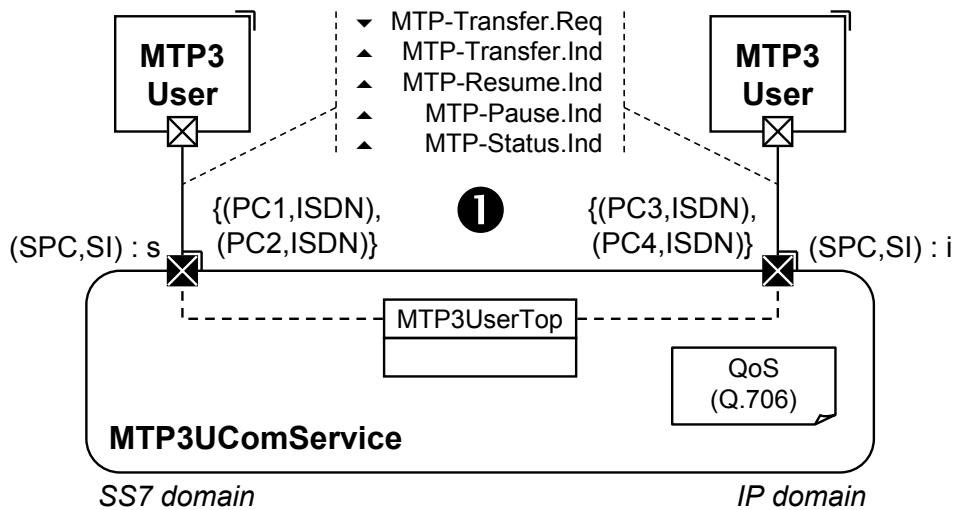


Figure 7.12: Model of the MTP3 user communication network

parameters, see [ITU96b]. Furthermore, three types of notifications can be delivered to the user: MTP-Pause.Ind, MTP-Resume.Ind, and MTP-Status. For the sake of clarity, the protocol messages have been annotated in figure 7.12; this is no standard ROOM(+++) notation.

Users of MTP3UComService are addressed via a Signaling Point Code (SPC) and a Service Indicator (SI). The SPC identifies the user's location, the SI the specific user part. In our model, we assume the user part to be ISUP (ISDN User Part) in correspondence to figure 1.3 b) (page 15) and figure 1.4 (page 16) in chapter 1. We assume the Point Codes (PC) PC1 and PC2 for users in the SS7 domain, see port s, and PC3 and PC4 for users in the IP domain, see port i. The destination of a message is called Destination Point Code (DPC); its origin is called Originating Point Code (OPC).

The model for MTP3UComService is an abstract one. The topology of "who can send messages to whom" is given by MTP3UserTop. The high quality demands of SS7 apply and are captured by the QoS note, which refers to [ITU93e]. SS7 services are extremely reliable and resistant against many sources of failure.

A complete specification of MTP3UComService in PyROOM++ is easy to realize. The simplicity and expressiveness of figure 7.12 contrasts to the figures of the SIGTRAN standard as shown in chapter 1. As a means to communicate the architecture vision of SIGTRAN, figure 7.12 already suffices.

Note that we neglected registration messages and configuration management services for the sake of simplicity.

7.2.3 Step 2: MTP3/M3UA Communication Network

Next, we substitute the abstract model of step 1 by a refined, more concrete model, which is – according to the recursive approach of our methodology – itself a communication network. Since the communicating entities on this level are protocol entities, namely MTP3 and M3UA, the corresponding method block applies.

If we ask ourselves “Which are the communicating entities within MTP3U-ComService (step 1)?”, SIGTRAN gives us a clear answer on it. Within the SS7 domain, MTP3 is the standard protocol entity to be taken [ITU96b]; within the IP domain, M3UA (MTP3 User Adaptation) is the recommended protocol entity [SMPB02], see figure 7.13.

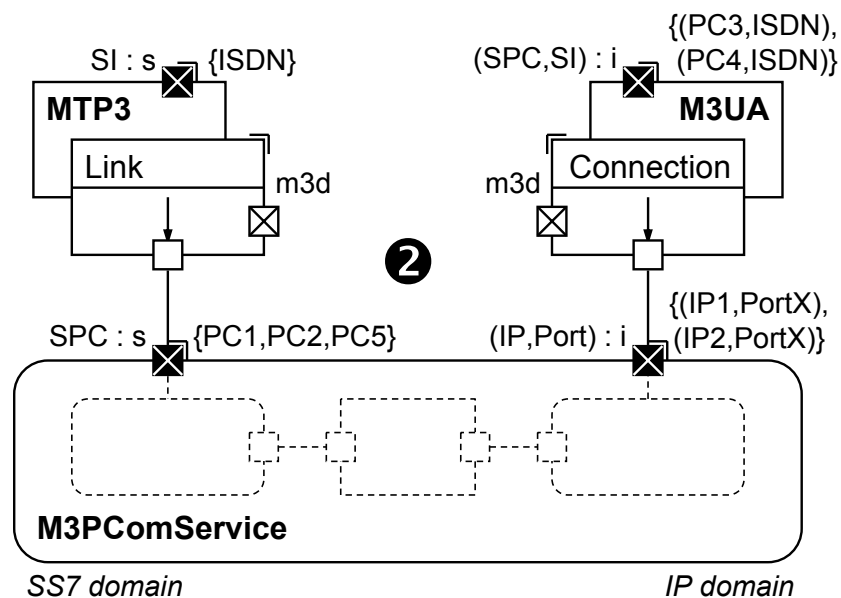


Figure 7.13: Model of the MTP3 provider communication network

MTP3 uses the notion of a Link for connection-oriented communication with its peers. It is pattern b) in figure 7.6 that comes into action: a connectionless communication service is realized via a protocol-oriented protocol. The address type on port *s* consists only of the SI, since the address type of the port MTP3 is attached to *i* is of type SPC. Remember, that address relaying (see chapter 5) ensures that this addressing scheme is consistent with the address scheme provided by the abstract model in step 1. With the support of our methodology, all this information can be retrieved from a detailed analysis of the MTP3 protocol as described in the standard, see [ITU96b]. The standard is also the resource to consult if one is interested in a refined model of MTP3. Briefly said, MTP3 can be decomposed in a Signalling Message Handling (SMH) part and a Signaling Network Manage-

ment (SNM) part; each part is further decomposed into three functional block, see figure 1 of [ITU96b].

Within the IP domain, SIGTRAN has specified the MTP3 User Adaptation layer (M3UA). M3UA provides a perfect MTP3-like interface and addressing scheme on port *i*. Internally, M3UA maintains the notion of a connection. Similar to links used by MTP3, M3UA uses connections to convey MTP3 user messages. In that sense, MTP3 and M3UA look very much alike. Both protocols use a sort of routing function to distribute messages on links and connections, respectively. The routing function is primarily based on the DPC and the OPC of an MTP3 user message. However, a closer look unveils fundamental technical differences: (a) M3UA connected to M3PComService changes the address space. Therefore, M3UA must have an address converter function that needs to be configured. Connections are addressed on their control port via an IP address and an IP port number, and these addresses must be somehow matched against SPC/SI pairs; (b) the IP/Port topology on the IP domain side is very different from the SPC topology on the SS7 side. More on the topology in step 3 and 4.

The abstract model of the communication service M3PComService is not complete. The dashed lines indicate that we decompose M3PComService into three entities: two communication services and one actor reference. The two communication services model the communication means within each domain, they are described in step 3 and 4; the actor reference models the glue between both communication technology domains, see step 5.

There is one important detail, one should not oversee. The address list attached to port *s* of M3PComService has an additional point code, PC5, which did not show up in figure 7.12. We listed PC5 as an example of an MTP3 entity, which has no user part attached to it. Consequently, we could not see PC5 in figure 7.12.

7.2.4 Step 3: MTP3 Communication Service

Figure 7.14 shows an abstract model of the communication service offered to MTP3 entities via port *s*. The model is derived from the connection-oriented communication pattern, see figure 7.3 a), and should be understandable without further comments. The actor references *partyA* and *partyB* can import MTP3 entities for raw message transfer via *m3d*. In SS7, links are usually pre-configured paths of communication, which are set-up prior to any message traffic.

Just for the sake of greater clarity, we split port *s* into two address “segments”. PC1, PC2 and PC5 represent point codes which are located in the SS7 domain, whereas PC6 belongs to the “grey zone” integrating the SS7 and IP technology domain.

An example link topology is given in the note box attached to LinkTopology. Since PC5 has no user part in our models, PC5 functions as a Signaling Transfer

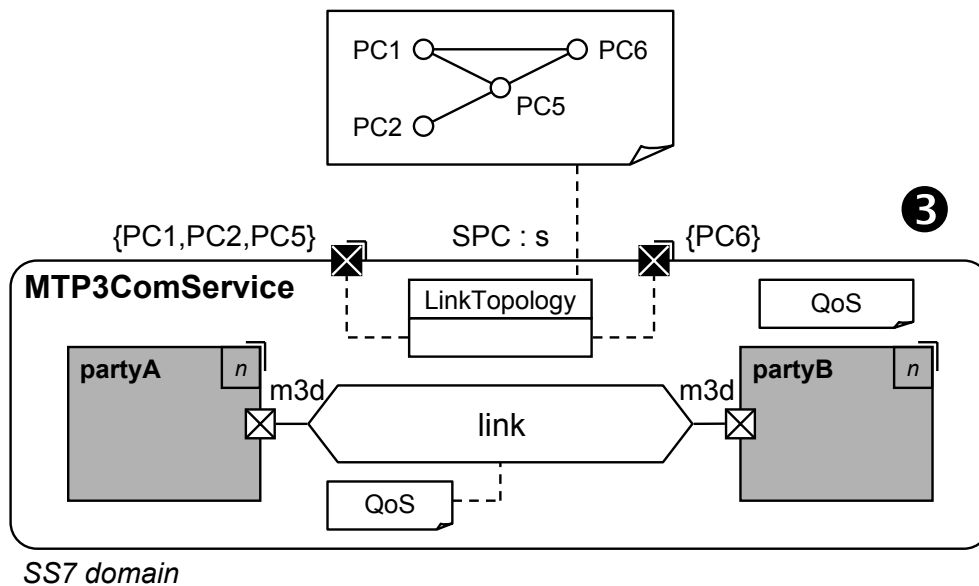


Figure 7.14: Model of the MTP3 communication service

Point (STP) only. If the routing functions of MTP3 are properly set up, the topology is resistant against failure of one link in the triangle spanned by PC1, PC5 and PC6.

Key to the understanding of SIGTRAN is that all messages targeted to PC3 and PC4 are routed to PC6. Also, all signaling messages originating from PC3 and PC4 will pop up at PC6 and take their route (a chain of links) to their destination, either PC1 or PC2. Point code PC6 is the point of interaction with the other technology domain.

7.2.5 Step 4: M3UA Communication Service

Figure 7.15 shows an abstract model of the communication service offered to M3UA entities via port *i*. Structurally, this model and the previous model (figure 7.14) are very similar and base on the same communication pattern. The actor references initiator and acceptor can import M3UA entities for the transfer of signaling messages via m3d. Also in M3UAComService, the sctpConnectors are usually pre-configured paths of communication. The prefix “sctp” stands for “Stream Control Transmission Protocol” (SCTP) [SXM⁺00], which is the protocol SIGTRAN recommends for the concretion of the sctpConnector.

As we did for the MTP3ComService model, we split port *i* into two address segments. Here, (IP3,PortX) belongs to the “grey zone” integrating the SS7 and IP technology domain.

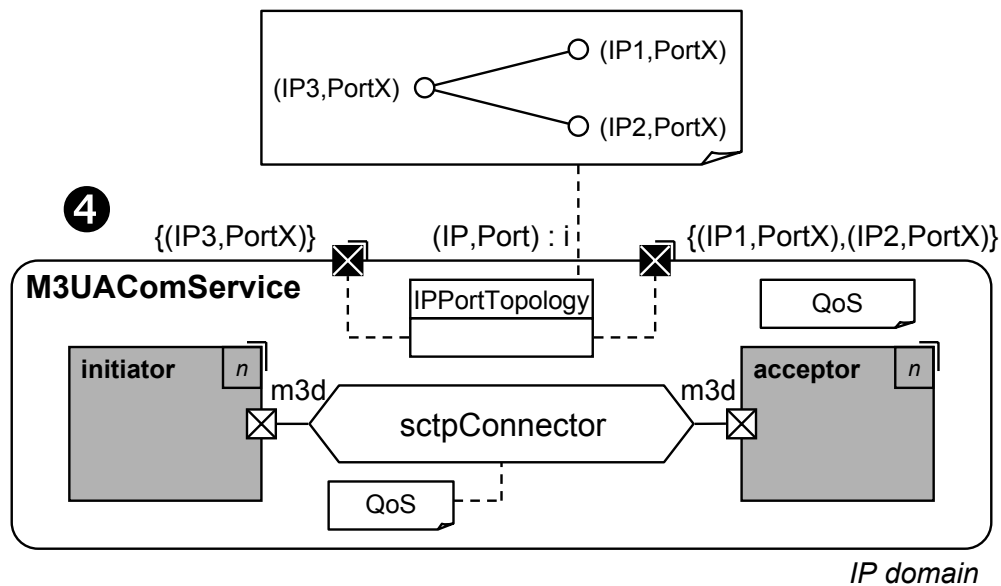


Figure 7.15: Model of the M3UA communication service

The address topology for M3UAComService looks different. In principle, there is no equivalent to STPs, all “endpoints” are directly connected to the interface point (IP3,PortX). Here, we drop the issue of how e.g. (IP1,PortX) and (IP2,PortX) can communicate to each other, since this is a delicate and not fully clarified matter in the M3UA standard. In general, the IPPortTopology is a star topology, whereas the LinkTopology (figure 7.14) can be – at least theoretically – complete but is irregular in practice. The reason for this categorical difference in the topology is that failures on the link level have to be compensated by alternative routes. However, in the IP domain, an sctpConnector is not assumed to be subject of failure, i.e. it is assumed to be very unlikely that a sctpConnector can break. Failures and appropriate measures to compensate them are delegated to lower protocols levels in the IP domain. If we would further decompose the M3UAComService we could more precisely point out, what is meant by that. Nonetheless, it is a controversy among experts in the field, whether these assumptions are appropriate and whether an sctpConnector requires a redundant connector in parallel or not. These few remarks may show that links and sctpConnectors are not that much comparable as it might seem at first sight. The philosophy and consequences regarding QoS (Quality of Service) are fundamentally different in SS7 networks and IP-based networks.

7.2.6 Step 5: Mediator

The actor class mediating between the MTP3ComService and the M3UAComService is called Mediator and shown in figure 7.16. The Mediator itself is modeled as a communication network with MTP* and M3UA* as the communicating entities and an InterworkingFunction as the communication service.

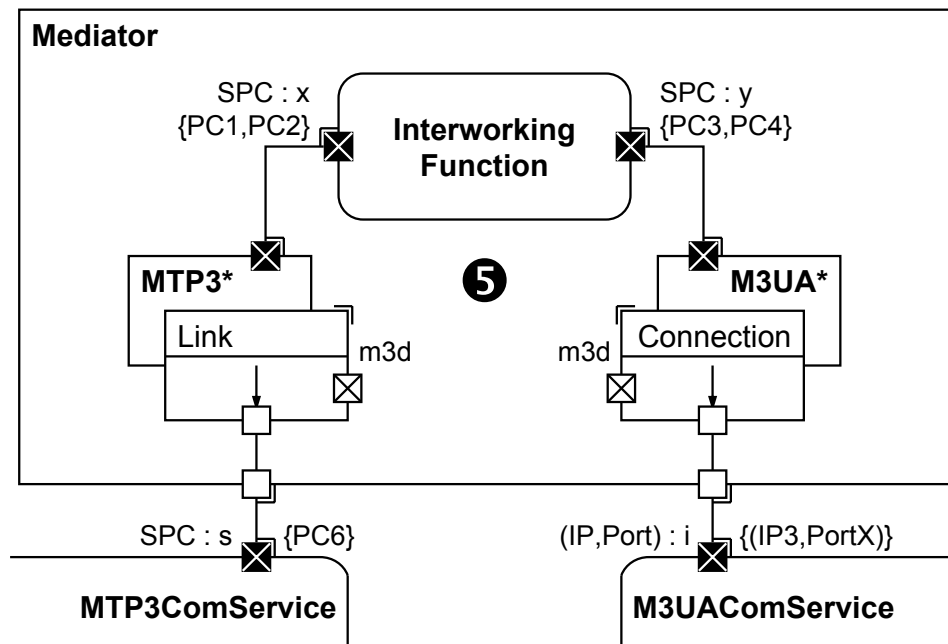


Figure 7.16: Model of the mediator

The star symbol attached to the actor class names MTP3* and M3UA* indicates that both actor classes are almost identical in functionality to MTP3 and M3UA, see figure 7.13. The subtle but important difference is explained for MTP. When a signaling message arrives at an MTP3 entity, the standard behavior is as follows: the message is either distributed to the user part or routed to another MTP3 entity. The criteria is whether the DPC of the signaling message matches the SPC address of the MTP3 entity or not. An MTP3* behaves exactly the same, but in addition, messages with certain SPCs can be handed over to the InterworkingFunction instead. In our case, messages that end up at PC6, which have PC3 or PC4 given as their DPC, are neither distributed nor routed but forwarded to the InterworkingFunction. A similar functional enhancement extends M3UA*.

The function of the InterworkingFunction is relatively trivial: it just delivers messages to the right port. A signaling message handed over by MTP3* is pushed to M3UA*, analyzed by M3UA* and finally delivered to its destination in the IP domain. If we would have taken network management functionality into account

as well, the InterworkingFunction would be far from trivial.

7.2.7 Step 6: Design View

We do not further refine our model on SIGTRAN, because the model developed so far contains everything SIGTRAN is basically all about. We could go on and further resolve the communication services MTP3ComService and M3UAComService; but we would not gain any insights that are specific to SIGTRAN.

The final question in our methodology is, which viewpoint we would like to take on the model we designed. Should it remain network-centric or should it be node-centric? Figure 7.17 shows both views in a single diagram. The diagram is not compliant to ROOM++ but rather a schematic sketch of the ROOM++ entities and their relations. This simplification should help not getting lost in too many details.

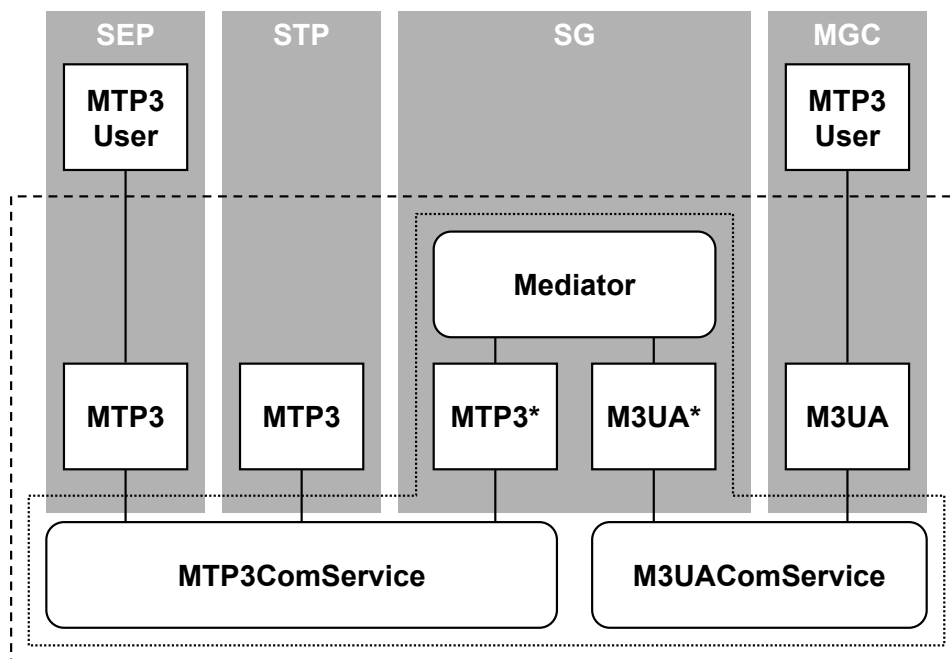


Figure 7.17: Views on the SIGTRAN model

The dashed line marks the borders of the MTP3ComService, the dotted line the borders of the M3UAComService; these lines show nesting levels of the network-centric viewpoint, which is also dominant in our methodology. The grey shaded areas mark a node-centric viewpoint on SIGTRAN. MTP3 users and MTP3 are hosted by nodes called *Signaling EndPoints* (SEPs), MTP3 entities without a user part are hosted by *Signaling Transfer Points* (STPs). On the right hand side, MTP3

users stacked upon M3UA reside in a node called *Media Gateway Controller*. The mediator including adapted versions of MTP3 and M3UA make up the *Signaling Gateway* (SG) node. From a node-centric viewpoint and on this level of abstraction, SEP, STP and SG communicate to each other via an MTP3 communication service; SG and MGC, on the other hand, communicate to each other via an M3UA communication service.

7.2.8 Step 7: Media Gateway

There is only one open item left: Where has the *Media Gateway* (MG) been? In figure 1.2 on page 13, the MGC is connected to the SG *and* the MG. Obviously, we have “forgotten” to model the MG. Figure 7.18 adds the missing piece to our model.

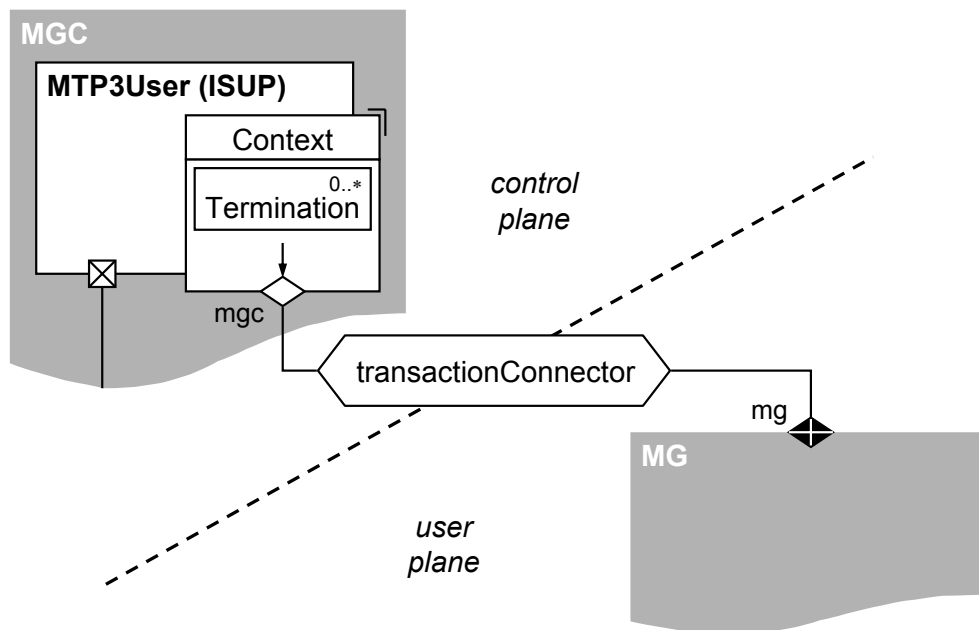


Figure 7.18: Modeling the MGC/MG relation

As a matter of fact, ISUP as our user part of choice does have an additional port *mgc* that lets ISUP control a resource, hosted by the MG node. The *transactionConnector* models the binding between ISUP and the controlled resource; we already discussed this kind of connector in chapter 5 and showed there also how the connector can be refined by an aspect entity.

It is vital to understand that the MGC and all the other nodes like the SG, the STP, and the SEP belong to a functional sphere called *control plane*. The control

plane is the signaling system. The MG, however, is part of another functional sphere, the *user plane*. Both planes interact only via dedicated interfaces. These interfaces can be distributed as shown in our example and are typically control-oriented. If we want, we can annotate the different planes by attaching a plane attribute to all ports of the control plane, and another plane attribute to all ports of the user plane.

7.2.9 Evaluation

In chapter 1 we criticized the standards and their kinds of architectural models. Let us briefly recap, in which way we improved in comparison to the standards.

- The “model diagrams” presented in the standards are rarely consistent, follow no clear conventions, and their interpretation is often vague and intuitive. The use of ROOM++ as a modeling language removes many of these deficiencies.
- In contrast to ROOM and many other modeling languages, ROOM++ has been extended to enable modeling of address spaces, planes, complex connectors, control models, and QoS. Addressing, for example, is a very important issue in communication architectures, which is usually dropped in architecture diagrams due to non-existing capabilities to express address types and address lists.
- The overall purpose of SIGTRAN is much more clearer than it has been before.
- In chapter 1 we complained about the insufficient expressiveness of protocol architectures. Our architecture model, on the opposite, is much more clearer about the protocol entities and their relations. We, for example, showed that M3UA and MTP3 are not that similar like conventional protocol architecture diagrams might suggest. We could also clarify the meaning and the functioning of the Nodal Interworking Function (NIF), see figure 1.4 on page 16.
- The notion of complex connectors (often declared as communication services in our diagrams) provided us with a very powerful abstraction technique. As our case study demonstrates, complex connectors enable the modeler to decompose a system in manageable model artifacts. We could also see that we can cut-off the refinement process at any suitable level.

The reader may go through the list of questions we raised in chapter 1 regarding SIGTRAN and judge him/herself how much we improved. We think that we

could considerably remove a lot of unclarities and substantially improve the quality and the expressiveness of architecture models for telecommunication systems.

7.3 Experiences

The methodology, the language ROOM++ and an early version of PyROOM++ have been developed during the authors employment at Ericsson and were tested under real conditions. With this background we can report some experiences we made with the practical use of our work.

Language

First was the development of ROOM++. The language extensions became relatively stable around mid of 2002, at least on a visual level, after a period of active development of over two years. The precise formalization of ROOM++ required another half a year.

ROOM++ turned out to be very helpful for developing an understanding of an architectural problem. The language proved its use for clarifying and communicating architectural designs. Several times, we experienced that the clarity gained by the use of ROOM++ released creativity and led to new solutions. In the case of SIGTRAN, two patents on SIGTRAN were the outcome.

We also learned that ROOM++ models, if explained once, got easily adopted by other system designers. However, understanding ROOM++ models and creating ROOM++ models is not the same. Without extensive training, there is not much of a chance that system designers use ROOM++ as a tool on their own initiative to express their architectural conceptions.

Interesting was also that engineers soon got bored by drawing the ROOM(++)-port symbols over and over again on the white board. Message schemata and port conjugation were rarely a matter of debate, so the graphical formalism was soon simplified.

Methodology

With the introduction of the complex connector concept to ROOM++, the language started to almost drive the methodology by itself. We continuously asked ourselves the questions what can be abstracted by a complex connector and what are the communicating entities. After some time, it was just a small step to work out the guidelines, heuristics and patterns we used over and over again. However, during the trial period at Ericsson, the difference between ROOM++ and the methodology was not that clear as it is now.

Tool

Except for small demonstration cases, PyROOM++ and its predecessor PyROOM were not used in the SIGTRAN case study. Even though PyROOM++ is simple to use for anybody familiar with the Python programming language, we experienced that many system designers do not necessarily need executable models. Often they have a quite well understanding of the general functioning of, say, a protocol

entity, so they just “emulate” a ROOM++ model in their minds. What seems to be much more important for system architects is to get a clear picture on the interwork of building components, how they relate to each other, how their address spaces are related, where and how Controlled Domain Models (CDM) play a role etc. In that respect, ROOM++ helps a lot since it provides many missing concepts required for such a structuring.

7.4 Summary

The methodology we presented in this chapter has a very generic approach: a *communication network* consists of *communicating entities* and a *communication service*, see figure 7.1. The communication service can be considered as a complex connector, which we introduced as a key modeling element to reason about (a) *distribution* in chapter 4 and (b) about *layering* in chapter 5. Consequently, we have two alternatives to express the communication service either by an *abstract model* or by a *concrete model*. If modeled abstractly, the communication network describes a distribution model. If modeled concretely, the communication network breaks down into another communication network – we called this form of layering more precisely *communication refinement*. Both models are interchangeable, since the service interfaces of the communication service remain not impacted by this approach.

The communicating entities can be peers, controllers or controllees. Depending on the type, we provided different kinds of method blocks that help analyze the situation. Very typical in communication system are *protocol entities* and *resources entities*. A generalization thereof is the *aspect entity*, which can be used to model anything else, applications or refinement aspects, as long as the constraints imposed by communication refinement are fulfilled.

The service interface provided by the communication service and used by communicating entities can be decomposed in “atomic” parts that separate control from data. According to the categorization in chapter 3, we can then distinguish between *control-oriented*, *protocol-oriented* and *data-oriented* types of communication. If control-oriented, the Controlled Domain Model (CDM) has to be defined. In addition, our method covers modeling address spaces and planes.

As a final example, we applied our methodology and ROOM++ on the SIGTRAN case study we introduced in the introduction of this work, see chapter 1. In comparison to the architecture diagrams presented by the SIGTRAN standard, we could drastically improve consistency and expressiveness of the architecture models.

To conclude, a methodology and a language adapted to the domain of telecommunication systems can team up to a strong and powerful engineering tool. The art is to find the right balance: a not too specialized language and a not too detailed methodology. We believe, we have found a good solution.

Chapter 8

Related Work

In this chapter we relate our approach to other published work. Problematic is that our investigation touches many fields of computer science. That is why we put our approach in a historical context and related it from there on to other work.

Section 8.1 provides the historical “environment” this work is an ancient of. Subsequent sections position ROOM to other languages (section 8.2), discuss recent approaches about modeling telecommunication systems (section 8.3), mention other frameworks (section 8.4), and reflect todays state-of-the-art in architecture modeling (section 8.5).

8.1 Historical Context

Modern telephony switching systems came to birth in the early 1960s. The new technology of computer control, called Stored Program Control (SPC), started to substitute electro-mechanical systems [MJ85]. One of the main advantages introducing SPC was flexible systems, where additions and changes could be introduced primarily through program modifications rather than through changes in the hardware [Vig64]. However, by the late '60s, it was time for a review. At Ericsson, one had learned that the current generation of SPC, as it existed in the late 1960's, was expensive and way too complex, with hindsight, for widespread use, except, to some extent, in the American Bell companies. The disadvantages were above all in the high costs of handling – design, testing, modification, fault-correction, production, installation, and operation and maintenance [MJ00]. What was needed was a new approach to structure and organize these complex systems. With the engineering techniques available at that time – “Structured Programming” is in the air [DDH72] –, the principle of functional modularity was a promising approach. Within Ericsson, it was IVAR JACOBSON, who made the important contribution of the “block concept” in 1967 [JCJÖ92], the structuring of the system into self-contained functional modules (blocks), with all interworking between blocks performed by software signals [Her99b]. The development of Ericsson's AXE switching system was based on these principles; it went into trial service late in 1976 and became and still is one of the most successful switching systems worldwide [MJ00].

Hand in hand with this development, the study of new languages was initiated. The industry was in need of languages highly adapted to the demands of programming and designing telecommunication systems. The outcome of these efforts were SDL (Specification and Description Language) [ITU99a], MSC (Message Sequence Chart) [ITU99c], CHILL (CCITT High Level Language) [ITU96c], and MML (Man-Machine Language) [ITU88]. All three languages have been standardized by CCITT (Comité Consultatif International de Télégraphique et Téléphonique) and are still in use today. In the early 1980's, SDL and MSCs were intended for system specification and design, CHILL for detailed design, coding and testing, MML primarily for operation and maintenance. Especially for coding, many companies developed their own variant of a programming language. For example, Ericsson developed PLEX (Programming Language for EXchanges) [Her99b], Northern Telecom PROTEL (PROcedure Oriented Type Enforcing Language) [PW82]; both languages are block structured.

The new complexity of switching systems by sheer size of code is impressive. Around 1980, several hundred programmers had produced over one million lines of code over a five years period for the DMS-100 switching system family of Northern Telecom. This represents over 15 000 procedures in 1500 mod-

ules [PW82].

From this viewpoint, it may come to no surprise that *architecture* is and always has been an important issue in telecommunication systems design. Architecture is and was a means to deal with complexity. Of course, the term “architecture” was not clearly defined, but absolutely in line with the design paradigm of that time: the modularization of a system is regarded as its architecture. Architectures were not modeled, as we tend to say today, but rather described either informally, usually in some sort of box-line diagrams, or formally with SDL. It is interesting to read, which design conceptions were identified for new software architectures in the 1980’s: independent subsystems for call control (features), signaling, and hardware control; data abstractions partitioned for each subsystem; formal communication protocols; concurrent and asynchronous operation of each subsystem; terminal-oriented control; layered virtual machines; FSM Specifications; application programs; systems programs [Law82] – the topicality of the list is astonishing.

Meanwhile, object-orientation conquered the world and became the dominating design and programming paradigm. Between 1989 and 1994, the number of object-oriented modeling languages and methods increased from less than 10 to more than 50 [Alh98]. New iterations of these first-generation methods and languages began to appear during the mid-1990s that incorporated each other’s techniques. Basically three popular approaches aimed for unification and standardization [Her99a], namely RUMBAUGH’S Object Modeling Technique (OMT) [RBL⁺91, Rum96], BOOCH’S Object-Oriented Design (OOD) method [Boo91, Boo94] and JACOBSON’S Object-Oriented Software Engineering (OOSE) method [JCJÖ92]. The outcome was the Unified Modeling Language (UML), which was adopted by the Object Management Group (OMG) for standardization in November 1997. Today, the UML is the de facto language for modeling and designing object-oriented systems. It is almost a knock-out criterion for a CASE (Computer Aided Software Engineering) tool vendor not to be UML compliant. Besides early and justified criticism from academia on the design and preciseness of the UML (see e.g. [SK98]), the UML has been a marketing success. The forthcoming version 2.0 of the UML will mark a new milestone in the evolution of the UML.

In parallel to the advances in object-oriented modeling, so-called Architecture Description Languages (ADLs) came onto the scene. Developers started to be dissatisfied with the practice of architectural design, which has been largely ad hoc, informal and idiosyncratic: architectural designs are often poorly understood, architectural choices are based more on default than solid engineering principles and architectural designs cannot be analyzed for consistency and completeness [GMW97]. In response, researchers in industry and academia developed a number of ADLs. Examples of ADLs include – among others – ACME [GMW97], Darwin [MDEK95], Rapide [LKA⁺95], Unicon [SDK⁺95] and Wright [All97].

Although all of these languages are concerned with architectural design, each favors a certain paradigm or application field. ACME and its XML-pedant ADML [ADM00] are an exception: they understand themselves as an interchange language for software architecture. Despite such efforts, there is no unification of ADLs in sight. Rather, UML 2.0 strives to evolve as an ADL in conjunction to being an OO modeling language. Since the OMG decided in 2001 to go for “Model Driven Architecture” (MDA) as their strategic direction [MM01], it is not unlikely that the UML will win the race as an ADL as well.

8.2 Position of ROOM to Related Languages

As was mentioned, the choice for ROOM was an *a priori* decision, i.e. ROOM was selected among other candidates as the preferred modeling language *before* we started our investigation on key design principles of telecommunication systems. We wanted to have a language at hand that (a) enables us to express architectural patterns and models of design and that (b) is close to the design culture to be found in the telecommunication domain.

ROOM claims to be suitable for the design of complex (real-time) systems. But how about so-called Architecture Description Languages (ADLs), which claim the very same (but usually leave out the attribute “real-time”)? What about SDL, another famous language used in telecommunications? And why not UML? In this subsection, we position ROOM as our preferred language of choice.

ROOM and Real-Time

Languages that target the description of real-time systems easily end up with concepts similar to ROOM. The encapsulation of independent threads of control and the need to synchronize the access to data often leads to the concept of a *component* (or actor, in ROOM) as an independent active object, a sort of port concept as a defined interface and a concept of pre-defined paths for communication. A very illuminating example is the book of MAGEE and KRAMER [MK99]. The authors explain important concepts and techniques in concurrent programming, abstractly in models and concretely in Java. The abstract modeling approach is based on an algebraic notation, a process calculus called Finite State Processes (FSP). The FSP belongs to the family of notations pioneered by MILNER [Mil89] and HOARE [Hoa85]. In the book, MAGEE and KRAMER soon start to visualize the models in a graphical language that reminds very much of ROOM.

ROOM as an ADL

On the other hand, modeling languages that target the high-level structure of the overall application rather than the implementation details must be capable to model *components*, *connectors* and their *configurations*; in addition, *tool support* for architecture development and evolution is a must. At least, this are the qualifying attributes of so-called Architecture Description Languages (ADLs) according to [MT00]. ROOM does fulfill the mentioned requirements and is also often categorized as an ADL if it combines its forces with the UML [RSRS99, GKMP⁺00].

ROOM vs. SDL

ROOM and SDL are two competing languages in the telecommunication area. Both languages are very close in spirit to design principles found for telecommunication systems, see [Her99b, EHS97]. And both languages are similar in design: they are component-based, support compositions of components, exchange messages and signals, respectively, for communication and have a virtual machine specifying the execution semantics. But which one to choose? We gave ROOM precedence over SDL because:

- SDL is not open to extensions. It has been a standardized language for decades. The language kernel is stable and will not undergo dramatic changes anymore. There is no extension mechanism built into the language.
- ROOM is not standardized, but ROOM's high-level features (actors, ports and bindings) are expected to be included in the forthcoming version 2.0 of the UML.¹ Taking UML's extension mechanisms into account [HvW99], the UMLified version of ROOM will be open to extensions and gain a lot of flexibility. This means that this work remains actual and on the spot since it can be uplifted to the most popular modeling language around. The choice for ROOM is – in that sense – an implicit choice for UML. However, a concrete study on that is left for further research. Problematic with UML 2.0 will be that it (most likely) will not have precise execution semantics. For a discussion of recent trends in UML standardization consult [SRK02, Dud02, Mel02, FT02, Dor02].
- Even if we forget about UML kind of extension mechanisms, the language design of ROOM as such is much more open to language changes and adaptations than SDL is. For example, ROOM's interface concept is slightly more advanced than the interface concept in SDL is, since ROOM defines a dedicated interface entity (port, SAP, SPP). If we think of extensions like adding behavior to ports, it is easy to do so in ROOM but a major effort in SDL; it is even against SDL's language design philosophy. Same is true for extensions to the binding concept. Since our need of language extensions was foreseeable, ROOM was clearly the preferred choice.
- Object-orientation (OO), the dominating modeling paradigm today, is an integral part of ROOM. ROOM fits much better to an OO paradigm than SDL does. For SDL, OO is a relatively recent addition to the language, see [ITU99a].

¹Status of information is April 2003. The UML 2.0 will just use another terminology.

- Behavior can be specified via state machine diagrams in ROOM. State machines have become one of the most common means to specify behavior. SDL's process diagrams are less favorite. However, we have to admit that behavior modeling is not our main concern. We could have taken a neutral position on that.
- SDL is a specialized niche language and will most likely always be a niche language despite efforts to ease the translation of SDL specifications to UML [ITU99b]. To this day, ROOM has been a niche language as well but with a promising future: most probably, ROOM will be integrated in the UML (see also remarks below).
- SDL is typically not regarded as an ADL. It is nobodies first choice for architecture modeling. This is maybe not an academic argument but important in the perception of people: ROOM keeps up with the state-of-art in architecture modeling.

ROOM and UML

We spoke about the future of ROOM and that ROOM most likely becomes a part of the UML. This development might give the impression that ROOM is inferior to UML. Despite its few basic language conceptions, ROOM is much more expressive than one might anticipate and in many aspects comparable to UML features: ROOM does know class diagrams for actors, protocols, and data objects. All associations between actor classes are resolved in form of contracts. UML's interface concept is close to the port concept (ports go beyond UML's interfaces). Aggregation and composition are both part of ROOM. Collaboration diagrams are supported by ROOM, state diagrams as well. Sequence diagrams are an option to ROOM.

In some sense, ROOM could be interpreted as a set of strict rules of how to use UML, or – to speak modern terminology – ROOM could be interpreted as an UML framework or profile. There is a white paper from SELIC and RUMBAUGH exactly trying to describe such a ROOM profile; this profile is usually referred to as UML-RT [SR98b, SR98a]. Note that UML-RT is not formally specified; it rather stands for the idea to adopt ROOM by UML. A different approach but with the same goal of integrating ROOM in UML is sketched e.g. in [RSRS99]; there, ROOM helps the UML to add features of an ADL. Shortly speaking, ROOM is not much less powerful in its expressiveness than UML is; it is just designed for a specific purpose with a specific domain in mind, for which the UML is not sufficiently staffed. The combination of ROOM and UML seems to be a beneficial next step for the evolution of both and is on its way with the forthcoming UML 2.0

Summary

To conclude, ROOM ranks among those modeling languages, which are regarded as appropriate for modeling software and/or system architectures. Insofar, ROOM is a solid ADL choice. In addition to that, ROOM has been designed for the domain of real-time systems: it supports a message-oriented communication paradigm (which is ideal for asynchronous, synchronous communication and remote communication), and decouples concurrent threads of execution. Finally, ROOM is very close in spirit to the design principles applied for telecommunication systems. All in all, ROOM is a good choice for our purpose. Though we would like to emphasize that ROOM is not the only possible choice. We took ROOM because of its elegance in design and ease of extensibility.

8.3 Modeling Telecommunication Systems

Latest since the UML has been standardized by the Object Management Group (OMG) and published in a series of book, *modeling* is on everybody's lips. Also the importance of the architecture level in software systems is more and more respected, see for example OMG's initiative on Model Driven Architecture (MDA) [MM01]. However, model-based *systems* engineering is not as developed and mature as model-based software engineering is [Fis98]. We already speculated that system engineers do not have much use of modeling languages as long as there is no guidance given of *how* to apply such languages in technical domains. Evidently, there has been published only little about that subject. When it comes to model telecommunication systems the fundus of literature is even smaller.

There are some few books, in which the object-oriented paradigm has been used to model communication systems. One example is "Object-Oriented Networks: Models for Architecture, Operations, and Management" [Bap94] from BAPAT. The book uses not only conventional object-oriented modeling concepts but also advanced concepts from specialization theory. The syntax used to capture the semantics of models is the Abstract Syntax Notation One (ASN.1) (see e.g. [Lar99]). BAPAT develops a classification scheme adapted to the needs of communication networks that enables a designer to develop understandable and meaningful object and class diagrams. The approach is descriptive and the techniques presented seem to be suited for modeling product architectures. The risk is that given "facts" are just schematically modeled (it is relatively easy to note down an object diagram for almost anything) without any reflection about the actual functioning and the actual meaning for the architecture. The actual value for architecture modeling and model simulation is highly questionable.

Another example is "Object-Oriented Network Protocols" from BOECKING [Boe00]. The book's intention is to provide a foundation for the object-oriented design and implementation of network communication protocols. While modeling of communication systems is not the topic of the book it is worth to have a look at the Modular Communication System (MCS) Framework developed by BOECKING. It gives an insight how protocols could be modeled and that object-orientation is a practical approach in protocol design.

A completely different approach is taken by "Modeling Telecom Networks and Systems Architecture: Conceptual Tools and Formal Methods", written by MUTH [Mut01]. This book condenses more of 20 years of experiences gained on the subject within Ericsson. It presents a method and a language for modeling telecommunication system and is based on the processing system paradigm [MHL01]. The whole field of communication systems is covered and a stringent methodology and classification scheme is discussed; nevertheless it remains unclear how all the pieces precisely fit together. Main criticism on the book is that the

language used is proprietary and not formally specified, so it is hard to verify if the approach is really consistent and works. Unfortunately, the author lacks a proof thereof. Nonetheless, MUTH's way of a viewing telecom systems was influential for this work.

8.4 Frameworks

Our work bases on the two most wide-spread frameworks in use for telecommunication and data communication systems, namely OSI RM and TCP/IP RM. Besides, there are some other frameworks around which address the topic of distributed communication system and propose a terminology, a set of conceptions, and a system architecture organization. The most important frameworks to mention are the Reference Model for Open Distributed Processing (RM-ODP) [Put01, ITU97], the Telecommunications Information Networking Architecture (TINA) [CM95], and the Object Management Architecture (OMA) [SS95a, SS95b], which is the basis for the Common Object Request Broker Architecture (CORBA) [MR97, COR02]. Basically, all three frameworks try to specify an environment that makes it as easy as possible to develop, install, and maintain distributed applications. In that sense, these frameworks go far beyond the scope of our work. However, we are not interested in a specific framework for a processing infrastructure but a method that enables us to design virtually any communication environment from the ground up, be it an RM-ODP, a TINA, a CORBA platform, or any other communication system. What we are after are, so to speak, “atomic” modeling concepts and a stringent methodology that helps us build any communication system or reference framework. It is left for further research to investigate the utilization of our approach for e.g. TINA or RM-ODP.

8.5 Architecture

We already mentioned that *architecture* was quite early an explicit issue in the design of telecommunication systems, see the remarks under “historical context”. Today, architecture is still regarded as a key instrument in managing the complexity of those systems, see e.g. [PKB⁺02]. Yet, it comes to some surprise that the term *architecture* has multiple definitions – no universal definition has been established. So far, the Software Engineering Institute (SEI) has collected more than 90 definitions from literature and practitioners.² Nonetheless, there is a trend to observe that

- architecture is largely regarded from a structural perspective; in addition, framework models, dynamic models and process models seem to be main perspectives [CBB⁺03, p.4]
- one can have different *views* on an architecture; a view emphasizes a certain aspect of an architecture. An influential paper on views is KRUCHTEN’S “4+1” approach to architecture [Kru95]
- a module-oriented viewtype and a component-and-connector (C&C) oriented viewtype are re-occurring themes, sometimes under different names, in literature; see, for example, [SG96, HNS00, Szy02]. A module tends to refer to a design-time entity, whereas a component tends to refer to a run-time entity [CBB⁺03, p.22]
- recurring forms within a viewtype have been widely observed; these forms are usually called architectural *styles*; a style is a specialization of element and relation types [CBB⁺03]
- design patterns [GHJV95] could be regarded as the finer-grained analog of architectural styles [CBB⁺03, p.33]. However, the borderline is not that clear to draw. More research is required on the relation of pattern-oriented software architecture (see e.g. [SSRB00]) and architecture modeling.

The viewtype we take with ROOM++ is clearly C&C: in ROOM++ we decompose any system into components (actors) and connectors (bindings). The C&C viewtype has a long-lasting tradition in the telecommunication domain (see SDL) and has been subject of research in the field of architecture modeling for years. While structural decomposition was mainly understood as component decomposition, the role of the connector has been neglected for quite some time. Meanwhile,

²See <http://www.sei.cmu.edu/ata> (2003-04-07)

the situation has changed: the connector has been discovered as a first-class modeling element, which requires as careful specification as a component does, see e.g. [SG96]. Connectors represent, for example, pipes or filters. Independent from our work, CLEMENTS et al. have also introduced the term *complex connector* with a similar meaning to our use of the term [CBB⁺03, p.113]. However, we found no evidence that anybody else has been developed such a stringent and methodological use of complex connectors as we do, including our sound algebraic reasoning. It just looks like that others start to discover the power of complex connectors in various domains such as in the area of aspect-oriented development, see [AK03]. What our work does not touch upon is how the C&C style of ROOM++ is related to other viewtypes like a module viewtype and an allocation viewtype. For UML-RT, there is a proposal in [Sel99].

Of course, there is a lot literature to find on the subject of (architecture) modeling of distributed systems in general, see e.g. [Rad01, Kle00], but these works are typically concerned with the application level of communication systems, which is just *one* aspect of a (tele)communication system. But telecommunication systems design is very much about a whole communication infrastructure, which is – spoken in modern terms – a highly specialized middleware. Drastically speaking, telecommunication design would be an easy field, if it were all about the application level.

Chapter 9

Summary and Outlook

This work closes with a short summary (section 9.1) and an outlook (section 9.2) on further research.

9.1 Summary

The aim of this work was to contribute to the goal to uplift architecture modeling of telecommunication systems to a mature discipline. We argued that an established, mature modeling discipline is characterized by the use of a suitable *modeling language*, the use of a *modeling tool*, and a systematic approach to transform problems (we focussed on system standards) to models, a *methodology*. We claimed to present a systematic approach to create logical models of network architectures of virtually any telecommunication system. The thesis statement was that such an approach can be based upon as few as three basic cornerstones: the *types of communication* and the design principles of *distribution* and *layering* in a network system.

9.1.1 Cornerstones

A very brief summary on the three cornerstones is given in the following:

Types of Communication

The question of control and the notion of a Controlled Domain Model (CDM) laid the reasoning for grey-box specifications and the basis for the distinction of three basic types of communication: *data-oriented communication* (zero-sided control), *control-oriented communication* (one-sided control), and *protocol-oriented communication* (two-sided control). The latter, protocol-oriented communication, proved to be a specialty of telecommunication systems. Architectural grey-box specifications reasoned on the aspect of control are an unique contribution of this work.

Distribution

To model distribution, the conception of a *complex connector* was identified as key. Components (actor classes in ROOM) and complex connectors are first class elements of our modeling approach and are sufficient to model any distributed communication system. Novel is the consequent and methodic use of the complex connector, see also the next item.

Layering

The “in-between” of components, namely the complex connector, also laid the basis for an improved understanding of layering in communication systems: *communication refinement* refines the complex connector into another distributed communication network. We could also generalize the notion of layering by the concept of stratifying network aspects.

9.1.2 Results

The outcome of our investigation is manifested by a modeling language, a tool implementing the language, and a method. The case study and industrial experience give good indications that the results achieved are of practical relevance and use.

Language

We took the Real-Time Object-Oriented Modeling (ROOM) language as a basis and extended and improved it step by step according to the insights gained on *types of communication, distribution and layering*. The following list summarizes the extensions and improvements that characterize ROOM++, the extended ROOM language:

- Unification of port/SPP/SUP concept
- Control and data ports
- Controlled Domain Models (CDM)
- Address types, address lists and address topologies
- Planes
- Relaxed port semantics
- Typed bindings
- Classification system

Tool

ROOM++ has been implemented in Python; that is why we called the tool PyROOM++. PyROOM++ is a full re-implementation of ROOM (with some minor deviations) plus all the above mentioned extensions. The meta-model of ROOM++ was streamlined in comparison to ROOM. Basic shortcomings of PyROOM++ are

- No GUI, i.e. there is no graphical modeling frontend available
- No formal behavior specifications as finite state machines; all behavior has to be specified via Python code

- Due to the use of Python, which is an interpreted language, PyROOM++ is not suited for real-time applications

These shortcomings characterize PyROOM++ as a research prototype, serving as a proof-of-concept implementation, and not as a tool for an industrial environment. Nonetheless, the use of Python promotes the idea of *rapid model prototyping*.

Method

We presented a systematic approach that guides a system architect in the process of transforming a standard to an architecture model. The method is divided in four method blocks:

- Method Block: System Network Architecture
- Method Block: Protocol Entity
- Method Block: Resource Entity
- Method Block: Aspect Entity

9.2 Outlook

Due to our experience with the ROOM++ notation in practice, we believe ROOM++ to be a serious proposal as an architecture modeling language. To be a competitive candidate, some deficiencies of ROOM++ need to be solved first: (a) There is no mapping given from logical models to physical entities (assignment viewtype). A good starting point in that direction is [Sel99], in which a mapping proposal is made for UML-RT (a variant of ROOM). (b) So far, ROOM++ follows a message-based communication paradigm. Letting components (actor classes) communicate by means of method calls, the option is given for an object-oriented communication paradigm on a component level. We also have to adapt the concept of a complex connector in that respect.

Another issue to investigate is how ROOM++ can be smoothly integrated with the forthcoming version of the UML, UML 2.0. The next UML release knows about components, ports, and connectors. However, a thorough analysis is required whether the UML 2.0 can be used much alike ROOM++ or if a specific UML 2.0 profile (a sort of language extension package) is required. The migration from ROOM++ to UML is an important strategic step to achieve acceptance in an industrial environment.

Further research is required on the idea to use communication refinement for the stepwise introduction of aspects in system models. Here, we see a huge potential for ROOM++ and for our methodology to contribute to recent advances in aspect-orientations, see e.g. [AK03].

Bibliography

- [3GT99] Customised Applications for Mobile network Enhanced Logic (CAMEL) Phase 3 – Stage 2. Technical Specification 3G TS 23.078 version 3.1.0, 3rd Generation Partnership Project, Valbonne, France, August 1999.
- [ADM00] The Open Group, 1010 El Camino Real, Suite 380, Menlo Park, CA 94025-4345, USA. *Architecture Description Markup Language (ADML), Version 1 (Document Number: I901)*, April 2000.
- [AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.
- [AK03] Colin Atkinson and Thomas Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, January/February 2003.
- [Alh98] Sinan Si Alhir. *UML in a Nutshell*. O’Reilly, 1998.
- [All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Technical report number: Cmu-cs-97-144, Carnegie Mellon University, May 1997.
- [AP98] V. S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer, 1998.
- [ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.
- [Bal96] Helmut Balzert. *Lehrbuch der Software-Technik: Software Entwicklung*. Spektrum Akademischer Verlag, 1996.
- [Bap94] Subodh Bapat. *Object-Oriented Networks – Models for Architecture, Operations, and Management*. Prentice Hall, 1994.

- [BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
- [BDH⁺98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What characterizes a (software) component? *Software – Concepts & Tools*, 19(1):49–56, 1998.
- [Ber99] Alfs Berztiss. Contexts, Domains, and Software. In P. Bouquet, L. Serafini, P. Breézillon, M. Beuerecetti, and F. Castellani, editors, *Modeling and Using Context; Second International and Interdisciplinary Conference, CONTEXT'99, Trento, Italy, September, 1999*, number 1688 in Lecture Notes in Artificial Intelligence, pages 443–446. Springer, 1999.
- [Boe00] Stefan Boecking. *Object-Oriented Network Protocols*. Addison-Wesley, 2000.
- [Boo91] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.
- [Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2nd edition, 1994.
- [BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, W3C, October 2000.
- [Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers. Standard RFC 1122, Internet Engineering Task Force, October 1989.
- [Bra96] John W. Brackett. Graduate Education in Software Engineering and Product-line Engineering. In *Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families, Las Navas del Marqués, Ávila, Spain; November 18–19, 1996*, November 1996.
- [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [Bro93] Manfred Broy. (Inter-)Action Refinement: The Easy Way. *Program Design Calculi, Series F: Computer and System Sciences*, 118, 1993.

- [Bro96] Manfred Broy. Towards a Mathematical Concept of a Component and Its Use. In *Componentware Users Conference 1996, Munich, Proceedings*. SIGS Publications, 1996.
- [Bro98a] Manfred Broy. Compositional Refinement of Interactive Systems Modelled by Relations. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, LNCS 1536, pages 130–149. Springer, 1998.
- [Bro98b] Manfred Broy. A uniform mathematical concept of a component: Appendix to [BDH⁺98]. *Software – Concepts & Tools*, 19(1):57–59, 1998.
- [BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.
- [BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, May 1969. This paper is the basic reference to the so-called Alternating Bit Protocol, even though this name has not been used in the paper itself.
- [CBB⁺03] Paul Clements, Felix Bachmann, Lenn Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Views and Beyond*. Addison-Wesley, 2003.
- [CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 3rd edition, 2001.
- [CGH⁺00] Fernando Cuervo, Nancy Greene, Christian Huitema, Abdallah Rayhan, Brian Rosen, and John Segers. Megaco Protocol Version 1.0. Standard RFC 3015, Internet Engineering Task Force, November 2000.
- [Cha94] Alan F. Chalmers. *Wege der Wissenschaft: Einführung in die Wissenschaftstheorie*. Springer, 3rd edition, 1994.
- [CM95] Martin Chapman and Stefano Montesi. Overall Concepts and Principles of TINA – Version 1.0. Tina baseline, TINA-C, February 1995.
- [COR02] Common Object Request Broker Architecture: Core Specification – Version 3.0. Specification formal/2002-11-03, Object Management Group (OMG), November 2002.

- [Cro82] David H. Crocker. Standard for the Format of ARPA Internet Text Messages. Standard RFC 822, Internet Engineering Task Force, August 1982.
- [DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and Charles A. R. Hoare. *Structured Programming*. Academic Press, 1972.
- [DeM78] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.
- [Dij68] Edsger W. Dijkstra. The structure of the “THE”-multiprogramming system. *Communications of the ACM (CACM)*, 11(5):341–346, May 1968.
- [DM95] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-oriented Programming: a Short Comparative Study. In *Proceedings of the Workshop on Reflection and Meta-Level Architectures and their Applications in AI, IJCAI’95*, pages 29–38, 1995.
- [Dor02] Dov Dori. Why Significant UML Change Is Unlikely. *Communications of the ACM (CACM)*, 43(11), November 2002.
- [Dud02] Keith Duddy. UML2 Must Enable a Family of Languages. *Communications of the ACM (CACM)*, 43(11), November 2002.
- [EHS97] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL – Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.
- [EV98] Jörg Eberspächer and Hans-Jörg Vögel. *GSM – Switching, Services and Protocols*. Wiley, 1998.
- [FGM⁺99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Standard RFC 2616, Internet Engineering Task Force, June 1999.
- [Fis98] Jerry Fisher. Model-Based Systems Engineering: A New Paradigm. *INSIGHT – A publication of the International Council of Systems Engineering (INCOSE)*, 1(3), 1998.
- [FT02] William Frank and Kevin P. Tyson. Be Clear, Clean, Concise. *Communications of the ACM (CACM)*, 43(11), November 2002.

- [Gel95] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1995.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [GKMP⁺00] David Garlan, John Knapman, Birger Møller-Pedersen, Bran Selic, and Thomas Weigert. Modeling of Architectures with UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *«UML» 2000 – The Unified Modeling Language: Advancing the Standard; Third International Conference, York, UK, October 2-6, 2000*, LNCS 1939. Springer, 2000.
- [GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997. IBM Center for Advanced Studies (CAS).
- [Gra95] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1995.
- [HABP02] Matt Holdrege, Ilya Akramovich, C. Michael Brown, and Bala Pitchandi. Megaco MIB. Internet Draft draft-ietf-megaco-mib-04.txt, Internet Engineering Task Force, October 2002. expires April 2003.
- [Hal96] Fred Halsall. *Data Communications, Computer Networks and Open Systems*. Electronic Systems Engineering Series. Addison-Wesley, 4th edition, 1996.
- [Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, pages 231–274, July 1987.
- [Her99a] Dominikus Herzberg. Die uml: Einführung und vergleich mit den konzepten und notationen von booch, rumbaugh and jacobson. Master's thesis, FernUniversität, Gesamthochschule in Hagen, February 1999.
- [Her99b] Dominikus Herzberg. UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain. In Robert France and Bernhard Rumpe, editors, *«UML» '99 – The*

- Unified Modeling Language: Beyond the Standard; Second International Conference, Fort Collins, CO, USA, October 28–30, 1999*, LNCS 1723, pages 330–338. Springer, 1999.
- [HM01] Dominikus Herzberg and André Marburger. The Use of Layers and Planes for Architectural Design of Communication Systems. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), Magdeburg, Germany; May 2-4, 2001*, pages 235–242. IEEE Computer Society, May 2001.
- [HMJ00] Dominikus Herzberg, André Marburger, and Tony Jokikyyny. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. 2. Workshop Software-Reengineering, Bad Honnef, Germany, 11.-12. May, 2000.
- [HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
- [Hoa85] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 1999.
- [HvW99] Dominikus Herzberg and Lars von Wedel. Erweiterungsmechanismen der UML. *OBJEKTSpektrum*, pages 56–59, Juli/August (4) 1999.
- [ITU88] Introduction to the CCITT man-machine language. ITU-T Recommendation Z.301, International Telecommunication Union, November 1988.
- [ITU91] B-ISDN Protocol Reference Model and its Application. ITU-T Recommendation I.321, International Telecommunication Union, April 1991.
- [ITU93a] ISDN Protocol Reference Model. ITU-T Recommendation I.320, International Telecommunication Union, November 1993.

- [ITU93b] Introduction to CCITT Signalling System No. 7. ITU-T Recommendation Q.700, International Telecommunication Union, March 1993.
- [ITU93c] Functional Description of the Message Transfer Part (MTP) of Signalling System No. 7. ITU-T Recommendation Q.701, International Telecommunication Union, March 1993.
- [ITU93d] Signalling network structure. ITU-T Recommendation Q.705, International Telecommunication Union, March 1993.
- [ITU93e] Signalling performance. ITU-T Recommendation Q.706, International Telecommunication Union, March 1993.
- [ITU93f] Information Technology – Open Systems Interconnection – Basic Reference Model: Conventions for the Definition of OSI Services. ITU-T Recommendation X.210, International Telecommunication Union, November 1993.
- [ITU93g] SDL Methodology Guidelines, SDL Bibliography. ITU-T Recommendation Z.100 Appendices I and II, International Telecommunication Union, March 1993.
- [ITU94] Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ITU-T Recommendation X.200, International Telecommunication Union, July 1994.
- [ITU96a] Signalling Link. ITU-T Recommendation Q.703, International Telecommunication Union, July 1996.
- [ITU96b] Signalling network functions and messages. ITU-T Recommendation Q.704, International Telecommunication Union, July 1996.
- [ITU96c] CCITT high level programming language (CHILL). ITU-T Recommendation Z.200, International Telecommunication Union, October 1996.
- [ITU97] Information technology – Open Distributed Processing – Reference model: Overview. ITU-T Recommendation X.901, International Telecommunication Union, 1997.
- [ITU99a] Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, November 1999.

- [ITU99b] SDL combined with UML. ITU-T Recommendation Z.109, International Telecommunication Union, November 1999.
- [ITU99c] Message Sequence Chart (MSC). ITU-T Recommendation Z.120, International Telecommunication Union, November 1999.
- [ITU00] Gateway Control Protocol. ITU-T Recommendation H.248, International Telecommunication Union, June 2000.
- [JBB92] Van Jacobson, Bob Braden, and Dave Borman. TCP Extensions for High Performance. Standard RFC 1323, Internet Engineering Task Force, May 1992.
- [JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering*. Addison-Wesley, revised printing edition, 1992.
- [KCe98] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised⁵ Report on the Algorithmic Language Scheme. <http://www.schemers.org>, February 1998. the report is the de facto scheme standard but not owned by a standardization body or any other organization.
- [Kes97] Srinivasan Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley, 1997.
- [Kle00] Peter Klein. *Architecture Modeling of Distributed and Concurrent Software Systems*. PhD thesis, Aachen University of Technology (RWTH Aachen), Germany, 2000.
- [Kle01] John C. Klensin. Simple Mail Transfer Protocol. Standard RFC 2821, Internet Engineering Task Force, April 2001.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer, June 1997.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.
- [Kru95] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

- [Lar99] John Larmouth. *ASN.1 Complete*. Morgan Kaufmann, 1999.
- [Law82] David A. Lawson. A New Software Architecture for Switching Systems. *IEEE Transactions on Communications Communication Software*, COM-30(6):17–25, June 1982.
- [LKA⁺95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.
- [Lut96] Mark Lutz. *Programming Python*. O’Reilly, 1996.
- [Lut01] Mark Lutz. *Programming Python*. O’Reilly, 2nd edition, 2001.
- [MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC ’95)*, number 989 in LNCS, pages 137–153. Springer, 1995.
- [Mel02] Stephen J. Mellor. Make Models Be Assets. *Communications of the ACM (CACM)*, 43(11), November 2002.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [MH01] André Marburger and Dominikus Herzberg. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. In Pedro Sousa and Jürgen Ebert, editors, *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001); 14-16 March 2001, Lisbon, Portugal*, pages 139–147. IEEE Computer Society Press, 2001.
- [MHL01] Thomas Muth, Dominikus Herzberg, and Jens Larsen. A Fresh View on Model-based Systems Engineering: The Processing System Paradigm. In *Proceedings of the 11th Annual International Symposium of The International Council on Systems Engineering (INCOSE 2001); July 1-5, 2001, Melbourne, Australia*, 2001.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [MJ85] John Meurling and Richard Jeans. *A Switch in Time – An Engineer’s Tale*. Telephony Publishing Corp., Chicago, Illinois 60604 USA, 1985.

- [MJ00] John Meurling and Richard Jeans. *The Ericsson Chronicle: 125 Years in Telecommunications*. Informationsförlaget Heimdahls AB, 11386 Stockholm, Sweden, 2000.
- [MK99] Jeff Magee and Jeff Kramer. *Concurrent Programming: State Models and Java Programs*. Wiley, 1999.
- [MM01] Joaquin Miller and Jishnu Mukerji. Model Driven Architecture (MDA). Technical Description ormsc/2001-07-01, Object Management Group (OMG), 2001.
- [Moc87a] P. Mockapetris. Domain Names – Concepts and Facilities. Standard RFC 1034, Internet Engineering Task Force, November 1987.
- [Moc87b] P. Mockapetris. Domain Names – Implementation and Specification. Standard RFC 1035, Internet Engineering Task Force, November 1987.
- [MR97] Thomas J. Mowbray and William A. Ruh. *Inside CORBA: Distributed Object Standards and Applications*. Addison-Wesley, 1997.
- [MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
- [Mut01] Thomas Muth. *Modeling Telecom Networks and Systems Architecture: Conceptual Tools and Formal Methods*. Springer, 2001.
- [Nag90] Manfred Nagl. *Softwaretechnik: Methodisches Programmieren im Großen*. Springer, 1990.
- [Nag96] Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Number 1170 in Lecture Notes in Computer Science. Springer, 1996.
- [Obj98] ObjecTime Limited, Canada, Ontario. *ObjecTime Developer: User Guide, Product Release 5.2*, August 1998.
- [OMG01] Unified Modeling Language Specification, Version 1.4. Technical Specification, Object Management Group (OMG), February 2001.
- [OMG02] Meta Object Facility (MOF) Specification, Version 1.4. Technical Specification, Object Management Group (OMG), April 2002.

- [Ora01] Andy Oram, editor. *Peer to Peer*. O'Reilly, 2001.
- [ORG⁺99] Lyndon Ong, Ian Rytina, Miguel-Angel Garcia, Hanns Juergen Schwarzbauer, Lode Coene, Hai an Paul Lin, Imre Juhasz, Matt Holdrege, and Chip Sharp. Framework Architecture for Signaling Transport. Standard RFC 2719, Internet Engineering Task Force, October 1999.
- [PC93] David M. Piscitello and A. Lyman Chapin. *Open Systems Networking: TCP/IP and OSI*. Addison-Wesley, 1993.
- [PD00] Larry L. Peterson and Bruce S. Davie. *Computer Networks – A Systems Approach*. Morgan Kaufman Publishers, 2nd edition, 2000.
- [PKB⁺02] A. Pink, H. Koßmann, M. Broy, E. Kargl, M. Lagally, and T. Schimper. *Software-Entwicklung für Kommunikationsnetze*. Springer, 2002.
- [Pop94] Karl Popper. *Logik der Forschung*. J.C.B. Mohr, 10th edition, 1994.
- [Pos80] Jon Postel. User Datagram Protocol. Standard RFC 768, Internet Engineering Task Force, August 1980.
- [Pos81a] Jon Postel. Internet Protocol. Standard RFC 791, Internet Engineering Task Force, September 1981.
- [Pos81b] Jon Postel. Transmission Control Protocol. Standard RFC 793, Internet Engineering Task Force, September 1981.
- [PR85] Jon Postel and J. K. Reynolds. File Transfer Protocol. Standard RFC 959, Internet Engineering Task Force, October 1985.
- [Put01] Janis R. Putman. *Architecting with RM-ODP*. Prentice Hall, 2001.
- [PW82] Brian K. Penny and J. W. J. Williams. The Software Architecture for a Large Telephone Switch. *IEEE Transactions on Communications Communication Software*, COM-30(6):105–114, June 1982.
- [Rad01] Ansgar Radermacher. *Tool Support for the distribution of Object-Based Applications*. PhD thesis, Aachen University of Technology (RWTH Aachen), Germany, 2001.
- [Rat00] Rational Software Corporation. *Modeling Language Guide, Rational Rose RealTime, Version 6.1*, January 2000.

- [Ray99] Eric S. Raymond. *The Cathedral & the Bazaar – Musings on Linux and Open Source by an Accidental Revolutionary*. O’Reilly, 1999.
- [RBL⁺91] James Rumbaugh, Michael Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [RSRS99] Bernhard Rumpe, Maurice Schoenmakers, Ansgar Radermacher, and Andy Schürr. UML + ROOM as a Standard ADL. In *Proceedings ICECCS’99, 5th International IEEE Conference on Engineering Complex Computer Systems*, pages 43–53, Los Alamitos, 1999. IEEE Computer Society Press.
- [Rum96] James Rumbaugh. *OMT Insights*. Prentice Hall, 1996.
- [Rus00] Travis Russell. *Telecommunications Protocols*. McGraw-Hill, 2nd edition, 2000.
- [S⁺96] John A. Stankovic et al. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763, December 1996.
- [SCFJ96] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. Standard RFC 1889, Internet Engineering Task Force, January 1996.
- [SDK⁺95] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.
- [Sel99] Bran Selic. Turning Clockwise: Using UML in the Real-Time Domain. *Communications of the ACM (CACM)*, 42(10):46–54, 1999.
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.
- [Sie97] Gerd Siegmund. *ATM – Die Technik: Grundlagen, Netze, Schnittstellen, Protokolle*. Hüthig, Heidelberg, 3rd edition, 1997.

- [Sim99] David E. Simon. *An Embedded Software Primer*. Addison-Wesley, 1999.
- [SK98] Martin Schader and Axel Korthaus, editors. *The Unified Modeling Language: Technical Aspects and Applications*. Physica-Verlag, Heidelberg, New York, 1998.
- [SM92] Sally Shlaer and Stephen Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1992.
- [SMPB02] Greg Sidebottom, Ken Morneault, and Javier Pastor-Balbas. Signaling System 7 (SS7) Message Transfer Part 3 (MTP3) – User Adaptation Layer (M3UA). Standard RFC 3332, Internet Engineering Task Force, September 2002.
- [Son98] Eduardo D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, 2nd edition, 1998.
- [Spu00] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. O’Reilly, 2000.
- [SR98a] Bran Selic and Jim Rumbaugh. Die Verwendung der UML für die Modellierung komplexer Echtzeitsysteme. *OBJEKTSpektrum*, pages 24–36, Juli/August 1998.
- [SR98b] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Whitepaper, Rational Software Corporation, March 1998.
- [SRK02] Bran Selic, Guus Ramackers, and Cris Kobryn. Evolution, Not Revolution. *Communications of the ACM (CACM)*, 43(11), November 2002.
- [SS95a] Richard Mark Soley and Christopher M. Stone. Object Management Architecture Guide – Revision 3.0. Document ab/97-05-05, Object Management Group (OMG), June 1995.
- [SS95b] Richard Mark Soley and Christopher M. Stone. *Object Management Architecture Guide, 3rd Edition*. Wiley, 1995.
- [SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

- [Sta88] John Stankovic. Misconceptions about real-time computing: A serious problem for next generation systems. *IEEE Computer*, 21(10):10–19, October 1988.
- [Sta96] John A. Stankovic. Real-time and embedded systems. *ACM Computing Surveys*, 28(1):205–208, March 1996.
- [Ste98] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI*. Prentice Hall PTR, 2nd edition, 1998.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.
- [Stu97] *Understanding Telecommunications 1*. Studentlitteratur, Lund, Sweden, 1997.
- [Stu98] *Understanding Telecommunications 2*. Studentlitteratur, Lund, Sweden, 1998.
- [SUN02] JavaSpaces Service Specification v1.2.1. Jini Specifications v1.2, Sun Microsystems, April 2002.
- [SX01] Randall R. Stewart and Qiaobing Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison-Wesley, 2001.
- [SXM⁺00] Randall R. Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns Juergen Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. Stream Control Transmission Protocol. Standard RFC 2960, Internet Engineering Task Force, October 2000.
- [Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.
- [Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 3rd edition, 1996.
- [Tan03] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 4th edition, 2003.
- [Tay98] Lloyd Taylor. Client/Server Frequently Asked Questions (Revision 1.12). <http://www.faqs.org/faqs/client-server-faq/>; FAQ of the comp.client-server newsgroup, August 1998.

- [Tuc97] Allen B. Tucker. *The Computer Science and Engineering Handbook*, chapter 78: Real-Time and Embedded Systems, pages 1709–1724. CRC Press, 1997.
- [TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey 07458, 2002.
- [Vig64] F. S. Viglinate. Fundamentals of stored program control of telephone switching systems. In *Proceedings of the 1964 19th ACM national conference*, pages 142.201–142.206, 1964.
- [Wal01] Bernhard Walke. *Mobile Radio Networks: Networking, Protocols and Traffic Performance*. Wiley, 2nd edition, 2001.
- [WAS01] Bernhard Walke, Marc Peter Althoff, and Peter Seidenberg. *UMTS – Ein Kurs*. J. Schlembach Fachverlag, 2001.
- [Wu98] Jie Wu. *Distributed System Design*. CRC Press, 1998.

Curriculum Vitae

■ Present Post

University of Applied Sciences, FH Heilbronn
since Apr. '03 **Stand-in Professorship** in Software Engineering

■ Previous Employment

Aachen University of Technology (RWTH Aachen)
Dep. of Computer Science III, Prof. Dr. M. Nagl
Dec. '02 – Mar. '03 **Research Assistant**, Architecture Modeling of
Telecommunication Systems
Ericsson Eurolab Deutschland GmbH, Herzogenrath
R&D for Mobile Circuit Switching Systems
Apr. '00 – Nov. '02 **Senior Systems Designer**
System and Product Management Department
Aug. '97 – Mar. '00 **Process Engineer**
System and Product Management Department
Jun. '95 – Jul. '97 **Test Engineer GSM**
Test Department

■ Education

Nov. '98 – Sep. '03 **External Dissertation**, RWTH Aachen
Oct. '96 – Mar. '99 **Wirtschaftsingenieurwesen**, FernUniversität Hagen
Oct. '88 – May '95 **Electrical Engineering**, RWTH Aachen
1986 – 1988 Civilian Service
1977 – 1986 Secondary school, Gymnasium Hückelhoven
1973 – 1977 Primary school, Hückelhoven

■ Personal Details

May 13, 1967 born in Bonn, Germany
nationality: German
married, one child

Aachen, September 2003

